Trabajo Fin de Grado:

PLATAFORMA INTEGRADA DE VISIÓN ESTÉREO PARA LA RECONSTRUCCIÓN 3D DE LA ESCENA CON PROCESAMIENTO PARALELO EN GPU

Autor: Diego Cabo Golvano

Tutor: Carlos Guindel Gomez

Grado en Ingeniería Electrónica, Industrial y Automática
Universidad Carlos III de Madrid
Septiembre, 2016

Acrónimos

- CPU: Computing Processor Unit (unidad de procesamiento central)
- GPU: Graphics Processor Unit (unidad de procesamiento gráfico)
- ISP: Image Signal Processor (procesador de señal de imagen)
- PCIe: Peripheral Component Interconnect express (interconexión de componentes periféricos)
- ROS: Robot Operating System (sistema operativo robótico)
- CUDA: Compute Unified Device Architecture (arquitectura unificada de dispositivos de cómputo)
- OpenCV: Open source Computer Vision (vision por computador de código abierto)
- SoC: System on a Chip (sistema en chip)
- IDE: Integrated Development Environment (entorno de desarrollo integrado)
- GPGPU: General-Purpose computing on Graphics Processing Units (computación de propósito general en unidades de procesamiento gráfico)
- UAV: Unmanned Aerial Vehicle (vehículo aéreo no tripulado)
- CAGR: Compound Annual Growth Rate (tasa de crecimiento anual compuesto)
- BSD: Berkeley Software Distribution (distribución de software de Berkeley)
- ADAS: Advanced Driver Assistance Systems (sistemas de asistencia a la conducción avanzados)
- IVVI: Intelligent Vehicle based on Visual Information (vehículo inteligente basado en información visual)
- LSI: Laboratorio de Sistemas Inteligentes

Tabla de figuras

Tabla de tablas

Agradecimientos

A mi madre, por aguantarme todos estos años y luchar por mi futuro sin tirar la toalla,

A mi padre, por ofrecerme su apoyo y consejo siempre que lo necesité,

A mi tío, cuyo ejemplo de investigación y esfuerzo en la universidad siempre será un modelo a seguir,

A mis compañeros de clase, quienes me ayudaron a hacer del día a día en la universidad más llevadero,

A mi tutor Carlos Guindel, ya que sin su ayuda este proyecto no habría salido adelante.

Abstract

This project presents a stereo vision algorithm CUDA-based GPU implementation on a small embedded system (NVIDIA Jetson TK1), which is capable of generating dense depth maps and 3D maps, close to a real-time operating mode. In order to free up some of the computational workload in the powerful computing unit of the Intelligent Systems Lab research platform IVVI 2.0, this system was designed as an independent module in a distributed computational platform approach. The different CPU and GPU OpenCV stereo algorithms will be studied, to see which fits best in this approach. Finally, a dense point cloud will be generated for each of the depth maps and used to generate a 3D map of the scene.

Resumen

En este proyecto se presenta la implementación en GPU basada en CUDA de un algoritmo de visión estéreo en un pequeño sistema embebido (NVIDIA Jetson TK1), el cual es capaz de generar mapas de profundidad densos y reconstrucciones tridimensionales de la escena, en un funcionamiento cercano al tiempo real. Con el objetivo de liberar parte de la carga computacional de la potente unidad de computación central de la plataforma de investigación IVVI 2.0 del Laboratorio de Sistemas Inteligentes, este sistema fue diseñado como un módulo independiente, en un enfoque de plataforma computacional distribuida. Los distintos algoritmos de visión en estéreo de las librerías OpenCV serán estudiados, para ver cuál es el que mejor se adapta a este objetivo. Finalmente, una nube de puntos densa será generada por cada mapa de profundidades y será utilizada para la creación de un mapa tridimensional de la escena.

1. Introducción

Hace algunas décadas, la introducción de la electrónica al mundo del motor produjo un salto cualitativo en la fabricación de vehículos. Motores más potentes y, a la vez, más eficientes y menos contaminantes. Consiguió que el mantenimiento del vehículo fuera mucho más fácil gracias a centralitas que indican el problema del coche, o simplemente recuerdan al conductor el cambio del aceite o de neumáticos.

Desde entonces, se han ido introduciendo nuevas mejoras para conseguir una conducción más cómoda y segura. Una de las líneas de investigación sobre la que más dinero se está invirtiendo en la actualidad es la conducción autónoma del vehículo. Gracias a múltiples sensores y potentes ordenadores integrados en el vehículo, hoy en día existen coches capaces de circular de forma autónoma por las carreteras con una gran seguridad.

Uno de los sistemas de ayuda a la conducción autónoma está basado en *'machine or computer vision'*, o visión por computador en español. Estos sistemas de visión por computador han avanzado enormemente en los últimos 30 años. Este avance se ha producido en gran medida gracias a librerías *'open source'* o de código abierto como OpenCV.

En este proyecto se abordarán dos puntos principales. El primero, la comparación entre algoritmos de visión en estéreo de las librerías OpenCV, ejecutados en la CPU, frente a los ejecutados en la GPU. Se analizarán datos como la velocidad de procesamiento, el error cometido, o la densidad de pixeles del resultado final. En la segunda parte, se realizará una reconstrucción en 3D de la escena, utilizando uno de los algoritmos de visión en estéreo en la GPU, y, sobre el meta sistema operativo ROS. La razón de esta primera parte del proyecto, se debe a que la calidad de la futura reconstrucción tridimensional, depende en gran medida del algoritmo de visión en estéreo utilizado.

Este trabajo, además, tiene como objetivo realizar esta reconstrucción en 3D utilizando el sistema embebido Jetson TK1. Un microprocesador con una potente GPU pensado para su uso en ámbitos de visión por computador y nubes de puntos, temas en donde la GPU adquiere una gran ventaja frente a la CPU gracias a su mayor capacidad de paralelización.

La decisión del uso de un sistema embebido no fue arbitraria. Hoy en día hay innumerables aplicaciones de visión por computador usadas en conducción autónoma de vehículos, drones, robótica, vigilancia… Pero no siempre existe la oportunidad de acceder a un potente ordenador para resolver la costosa carga computacional de estos algoritmos de visión en estéreo. Hay múltiples ocasiones en el que el peso, espacio y consumo sufren de grandes restricciones, como puede ser el caso de UAVs o robots de tamaño medio. No obstante, el uso de estos sistemas embebidos de bajo consumo se puede extender a los vehículos autónomos, aliviando la carga de trabajo a las unidades de procesamiento centrales.

Finalmente se obtendrá un sistema compacto y de bajo consumo, capaz de proporcionar visión y auto-localización espacial a sistemas inteligentes.

1.1. Problemática actual

En base a un estudio realizado por la Organización Mundial de la Salud [1], se estima que, cada año, mueren en el mundo 1,2 millones de personas en accidentes de tráfico, y hasta 50 millones resultan heridas. Esto es más de 3000 personas que mueren cada día por lesiones resultantes del tráfico. En los países de ingresos bajos y medianos se concentra aproximadamente un 85% de esas muertes y el 90% de la cifra anual de años de vida ajustados en función de la discapacidad (AVAD) perdidos por causa de esas lesiones. Las proyecciones muestran que, entre 2000 y 2020, las muertes resultantes del tránsito descenderán en torno al 30% en los países de ingresos altos, pero aumentarán considerablemente en los de ingresos bajos y medianos. De no emprenderse las acciones pertinentes, se prevé que, en 2020, las lesiones causadas por el tránsito sean el tercer responsable de la carga mundial de morbilidad y lesiones.

En la Figura 1 se observa el cambio en las diez causas principales de morbilidad según los AVAD perdidos. Claramente las muertes producidas en accidentes de tráfico han escalado en gran medida por esta tabla, situándose en 3er lugar.

Cambio de rango de las diez causas principales de la carga mundial de morbilidad según los AVAD perdidos

1990		2020	
Rango	Enfermedades o traumatismos	Rango	Enfermedades o traumatismos
1	Infecciones de las vías respiratorias inferiores	1	Cardiopatía isquémica
2	Enfermedades diarreicas	2	Depresión unipolar grave
3	Trastornos perinatales	3	Traumatismos causados por el tránsito
4	Depresión unipolar grave	4	Trastornos cerebrovasculares
5	Cardiopatía isquémica	5	Enfermedad pulmonar obstructiva crónica
6	Trastornos cerebrovasculares	6	Infecciones de las vías respiratorias inferiores
7	Tuberculosis	7	Tuberculosis
8	Sarampión	8	Guerras
9	Traumatismos causados por el tránsito	9	Enfermedades diarreicas
10	Anomalías congénitas	10	VIH

AVAD: años de vida ajustados en función de la discapacidad. Medición del desequilibrio en salud que combina información sobre el número de años perdidos por muerte prematura y la pérdida de salud por discapacidad.

Figura 1: Cambio de rango de las diez causas principales de la carga mundial de morbilidad [1]

Todas las personas que mueren, se lesionan o quedan discapacitadas por un choque tienen una red de personas allegadas, como familiares y amigos, que resultan profundamente afectadas. En el mundo, millones de personas se enfrentan a la muerte o la discapacidad de familiares debido a lesiones causadas por el tránsito. Sería imposible asignar un valor cuantitativo a cada caso de sacrificio y sufrimiento humano, sumarlos todos y obtener una cifra que refleje el costo social mundial de los choques y las lesiones causadas por el tránsito. Se estima que el costo económico de los choques y las lesiones causadas por el tránsito asciende al 1% del producto nacional bruto (PNB) en los países de ingresos bajos, al 1,5% en los de ingresos medianos y al 2% en los de ingresos altos. El costo mundial se estima en US$ 518 000 millones anuales, de los cuales US$ 65 000 millones corresponden a los países de ingresos bajos y medianos; este monto es mayor del que reciben en ayuda al desarrollo.

Las lesiones causadas por el transito representan una pesada carga no sólo para la economía mundial y de los países, sino también para la de los hogares. La pérdida de quienes ganaban el sustento y el costo añadido de atender a los familiares discapacitados por dichas lesiones sumen a muchas familias en la pobreza.

Estimación de los fondos destinados mundialmente a investigación y desarrollo sobre determinadas cuestiones

Enfermedad o traumatismo	Millones de US$	Rango según los AVAD, 1990	Rango según los AVAD, 2020
VIH/SIDA	919–985	2	10
Paludismo	60	8	–
Enfermedades diarreicas	32	4	9
Traumatismos causados por el tránsito	24–33	9	3
Tuberculosis	19–33	–	7

Figura 2: Estimación de los fondos destinados mundialmente a investigación y desarrollo sobre determinadas cuestiones [1]

En cambio, se invierte muy poco dinero en prevenir los choques y las lesiones causadas por el tránsito. En la Figura 2 se comparan los fondos gastados en labores de investigación y desarrollo centradas en varios problemas de salud, incluida la seguridad vial. Se gasta relativamente poco en la aplicación de medidas, pese a que muchas intervenciones que permiten prevenir choques y traumatismos se conocen bien, se han probado lo suficiente y son rentables y aceptables para la población.

1.1.1. Cifras en España

La mayoría de los accidentes que se producen anualmente en nuestro país ocasionan únicamente daños materiales originando importantes pérdidas económicas. Sin embargo, por su trascendencia para la salud de la población lo que resulta fundamental es conocer el número de accidentes con alguna víctima, las características en relación a la gravedad de las lesiones y los factores que desencadenan los accidentes.

Según el estudio de las principales cifras de la siniestralidad vial en España 2014 [2], las diferentes policías notificaron 91.570 accidentes con víctimas. Estos accidentes ocasionaron 1.688 fallecidos en el momento del accidente o hasta 30 días después del mismo, 9.578 personas fueron ingresadas en un centro hospitalario y 117.058 resultaron heridos no hospitalizados, según fuentes policiales. Estas cifras, aun siendo elevadas, han supuesto una estabilización con respecto al año anterior en el número de fallecidos (0%), una reducción del número de heridos hospitalizados (-5%) y un aumento en los accidentes con víctimas y en el número de heridos no hospitalizados (2%).

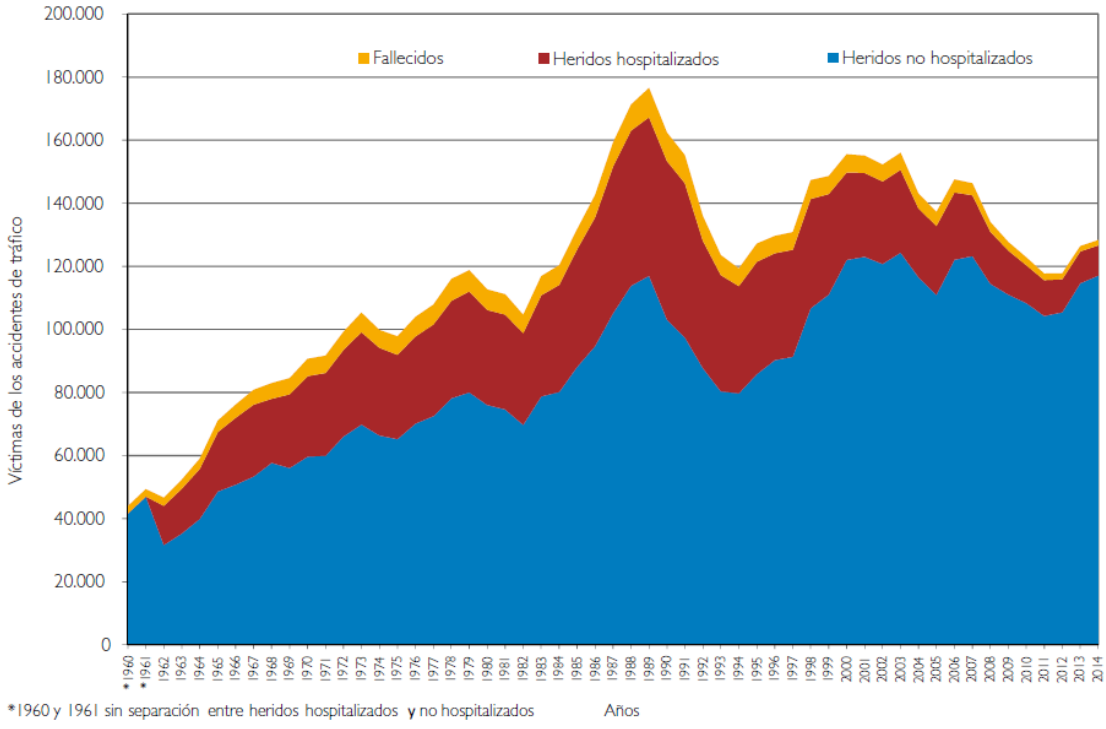


Figura 3: Evolución de las víctimas de accidentes de tráfico. España, 1960-2014 [2]

En la Figura 3 se observa la evolución de las víctimas de accidentes de tráfico en España desde los años 1960-2014. Mientras que el número de heridos no hospitalizados ha seguido aumentando en los últimos años, debido al mayor número de coches en circulación, se aprecia que el número de heridos hospitalizados y fallecidos se ha reducido significativamente. Esto es debido a la modernización de la red de carreteras, las nuevas medidas adoptadas, y a la multitud de mejoras de seguridad introducidas en los nuevos vehículos. Es en este último campo donde

una hipotética futura implantación de tecnologías similares a las de este proyecto, podría conducir a una mejora considerable en la tasa de siniestralidad.

Entre los factores que **inciden en la seguridad** cabe destacar que la **velocidad** inadecuada estuvo presente en el 10% de los accidentes con víctimas y que este porcentaje se eleva al 17% cuando el accidente con víctimas sucede en vía interurbana. En el caso especial de los accidentes con víctimas mortales se observa que este factor estuvo presente en el 21% de los accidentes.

Conducir después del consumo de **sustancias psicoactivas** es, desafortunadamente, un hecho frecuente en España, el 12% de los españoles que conducen turismo han consumido alguna droga de comercio ilegal y/o alcohol, antes de conducir. El 39% de los conductores fallecidos presentaron alguna sustancia psicoactiva, siendo el alcohol en el 67% de los casos, drogas de comercio ilegal en el 34% y psicofármacos en el 27%, según el informe del Instituto Nacional de Toxicología y Ciencias Forenses. Este informe también determina que el 40% de los peatones presentaron resultados positivos a drogas y/o psicofármacos y/o alcohol.

La **distracción** aparece como un factor concurrente en un 30% de los accidentes con víctimas, siendo esta proporción del 36% en las vías interurbanas y 26% en las urbanas. Uno de los motivos que produce la distracción es la utilización del teléfono móvil durante la conducción. En el año 2014 la Agrupación de Tráfico de la Guardia Civil formuló 135.852 denuncias por utilización del teléfono móvil, un 7% más que en el año 2013.

Las **infracciones**, en el año 2014, al menos el 44% de los conductores implicados en accidentes ocurridos en vía interurbana había cometido alguna infracción, un 3% menor que el observado el año anterior. En vía urbana, este porcentaje fue del 56%, un 6% menor que el año anterior. Al comparar los infractores e infractores reincidentes en población general y en conductores implicados en un accidente, se observa una proporción mayor de infractores e infractores reincidentes en los conductores implicados en accidentes, siendo esta del 11%.

Indicadores	Cifra basal 2009	Cifra 2013	Cifra 2014	Cifra objetivo 2020
1. Bajar la tasa de fallecidos a 37 por millón de habitantes	59	36	36	Inferior a 37
2. Reducción del número de heridos graves en un 35% [2]	13.923	10.086	9.574	9.050
3. Cero niños fallecidos sin sistema retención infantil [3]	12	4	2	0
4. 25% menos conductores de 18 a 24 fallecidos y heridos graves en fin de semana	730	345	360	548
5. 10% menos de conductores fallecidos mayores de 64 años	203	182	213	183
6. 30% reducción de fallecidos por atropello	459	349	310	321
7. 1 millón de ciclistas más sin que se incremente su tasa de mortalidad	1,2	1,5	1,6	1,2
8. Cero fallecidos en turismos en zona urbana	101	72	71	0
9. 20% menos de fallecidos y heridos graves de usuarios de motocicleta	3.473	2.811	2.870	2.778
10. 30% menos de fallecidos por salida de vía en carretera convencional	520	285	277	364
11. 30% menos de fallecidos *in itinere*	170	100	99	119
12. Bajar del 1% los positivos en aire espirado en los controles preventivos aleatorios. DRUID, punto de corte 0,05 mg/l	6,7%	4,1%	No disponible. Estudio bienal	Inferior al 1%
13. Reducir en 50% el % de vehículos ligeros que superan el límite de velocidad en más de 20 km/hora	12,3% (autop.) 6,9% (autov.) 15,8% (conv. 90) 16,4% (conv. 100)	No disponible. Estudio periódico	No disponible. Estudio periódico	6,2% (autop.) 3,5% (autov.) 7,9% (conv. 90) 8,2% (conv. 100)

[2] En los indicadores 2, 4 y 9, se entiende por herido grave toda persona herida en un accidente de circulación cuyo estado precisa una hospitalización superior a veinticuatro horas.
[3] Niños de menos de 12 años.

Figura 4: Indicadores de la Estrategia de Seguridad Vial 2011-2020. España, 2009,2013, 2014, 2020 [2]

La Figura 4 muestra la tendencia y estrategia de mejora de la seguridad vial para el año 2020. Se puede observar que muchos de estos objetivos pueden ser cumplidos con la ayuda de trabajos en el campo de visión por computador como el de este proyecto, u otros trabajos realizados en el departamento.

1.1.2. Tendencia en el mercado de cámaras 3D

La cuota de mercado de las cámaras 3D no se limita únicamente al mundo de la automoción. La introducción de estas cámaras ha revolucionado el mundo de la imagen digital. Áreas tan diversas como los videojuegos, películas 3D, monitorización de actividades industriales, domótica, monitorización de áreas públicas para vigilancia y otras están adoptando el uso de cámaras 3D para sus aplicaciones. Es más, el avance en algoritmos de escaneado 3D ha permitido el desarrollo de nuevos y mejores productos basados en cámaras 3D. La introducción de Smartphone y tablets habilitadas con tecnología 3D (como por ejemplo el proyecto TANGO entre Google y Lenovo [3]) también está alimentando el crecimiento de esta tecnología.

Este incremento del uso de contenido 3D generara un aumento en el mercado de las cámaras 3D. Sin embargo, los altos precios de las cámaras 3D limitan el crecimiento de este mercado. Además, como contrapartida, la facilidad al acceso a productos 2D limita también el crecimiento del mercado. En la Figura 5 se observan los factores más importantes que

influencian el uso de cámaras 3D. El mercado está mostrando la entrada de nuevos competidores debido al aumento en campos de domótica y realidad virtual, con cámaras cada vez más accesibles para el público general.

El estudio realizado por P&S Market Research [4] segrega el mercado global de camaras 3D en base a tipo, aplicaciones, tecnología y geografía. Las tecnologías usadas en camaras 3D son tiempo de vuelo (*time of flight*), visión estéreo (*stereo vision*) y escáner de luz estructurada (*structured light imaging*). Los precios inferiores de las cámaras estéreo y la facilidad de instalación de tecnologías de visión estéreo, han atraído a los fabricantes a usar esta tecnología, convirtiéndola en líder del mercado de cámaras 3D con más del 60% en 2015.

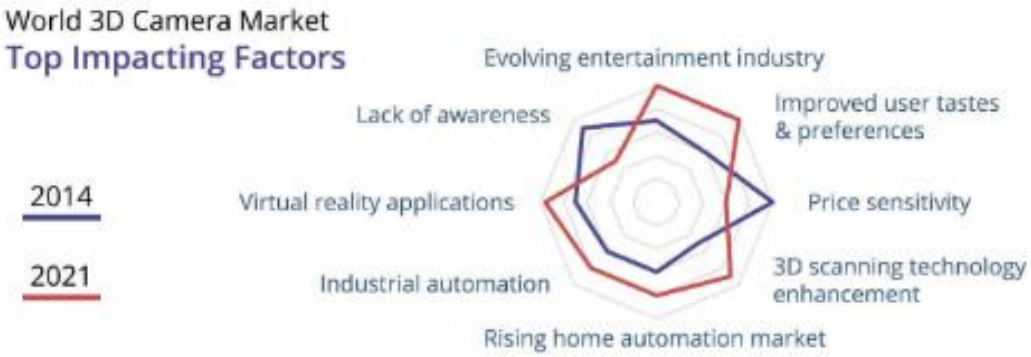


Figura 5: Factores más importantes en el mercado de cámaras [5]

En la Figura 6 se puede observar la tendencia de incremento en el mercado global de cámaras 3D profesionales. Éste mercado tiene una previsión de crecimiento CAGR (Compound Annual Growth Rate) del 50.2% durante el periodo de 2016-2022 (Figura 7), registrando unos beneficios totales de unos 19.893,0 millones de dólares para 2022. Los productos cada vez más avanzados tecnológicamente y el aumento del uso en smartphones potenciara aún más este crecimiento. El desarrollo de economías como la China o India está presenciando el mayor crecimiento en este tipo de cámaras. Tareas como monitorización remota, domótica y vigilancia son áreas en donde se observará un gran crecimiento en el uso de cámaras 3D durante el periodo previsto.

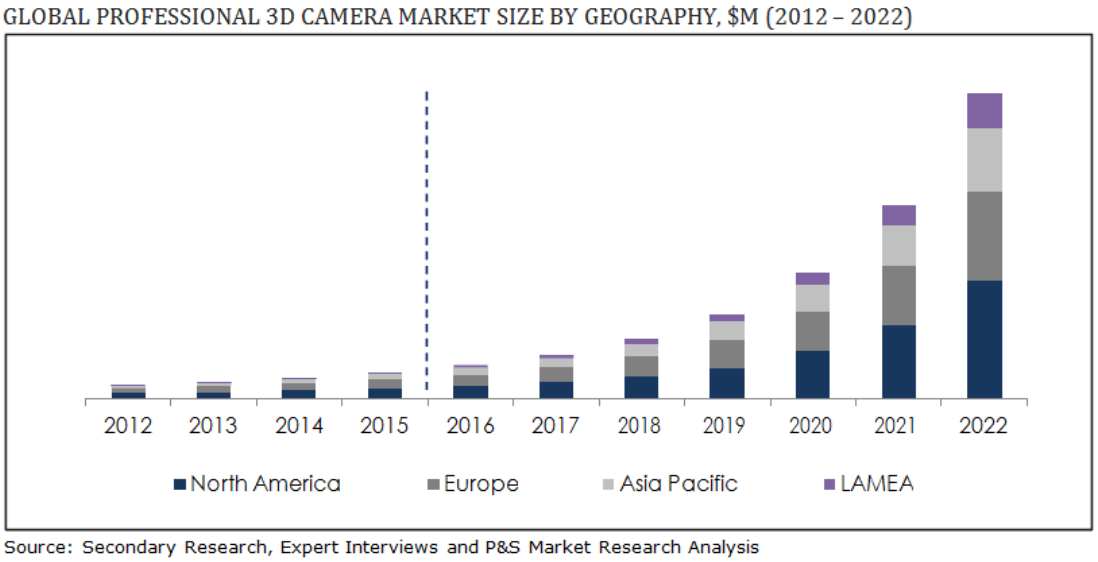


Figura 6: Incremento en el mercado de cámaras 3D profesionales segmentado por geografía [4]

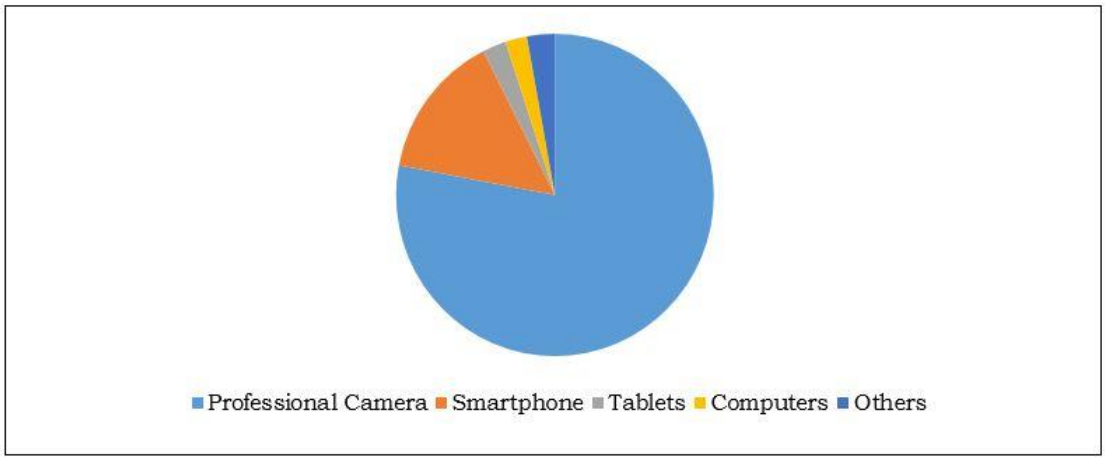


Figura 7: Beneficios en el mercado de cámaras 3D por aplicación. (% Comparación), 2014 [5]

Un dato muy significativo es el número de cámaras Microsoft Kinect vendidas en todo el mundo. Más de 24 millones de unidades han sido vendidas desde su lanzamiento en EE.UU. el 4 de noviembre hasta febrero del 2013 [6], habiendo sido vendidas 8 millones de estas unidades en los primeros 60 días desde su salida al mercado (Figura 8). Con estas cifras, Kinect ostenta el Record mundial Guinness por ser el aparato de electrónica de consumo más vendido [7].

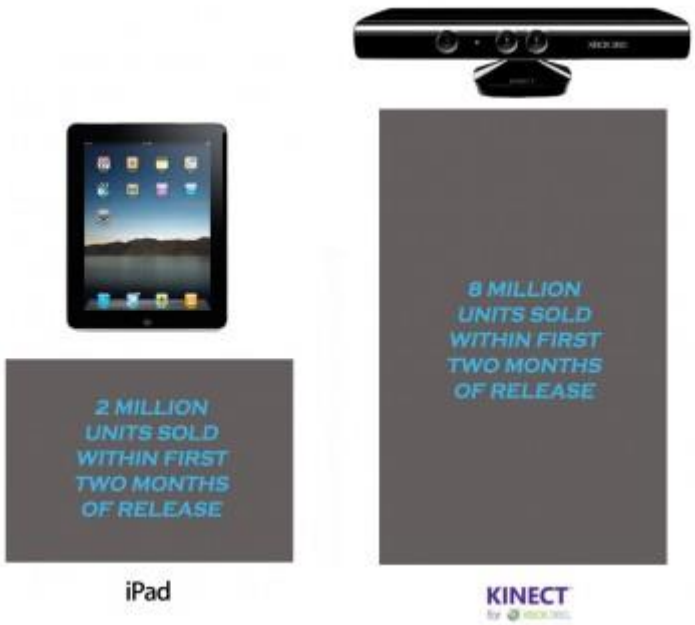


Figura 8: Comparativa entre ventas en los primeros dos meses de iPad y Microsoft Kinect [8]

1.2. Objetivos

Este proyecto tiene como objetivo inicial la creación de una aplicación de mapeado tridimensional basada en visión estéreo para una plataforma embebida (Jetson TK1).

Esta aplicación debe estar enfocada con una filosofía de computación distribuida. Actualmente, en el vehículo de investigación del LSI-Laboratorio de Sistemas Inteligentes IVVI (Intelligent Vehicle based on Visual Information), todo el procesamiento se realiza en el mismo equipo. La idea inicial fue la de liberar en cierta medida la carga computacional de este potente equipo central, dividiendo las diferentes tareas en módulos más o menos independientes, que se puedan ejecutar en equipos auxiliares.

No obstante, se debe dividir este objetivo general en otros puntos u objetivos más concretos, en los que se centra el estudio de este trabajo, consiguiendo así una mejor compresión de lo realizado en el proyecto.

- Aprovechamiento de las capacidades GPGPU de la Jetson TK1 para el procesamiento de datos en paralelo, en las partes criticas (mayor carga computacional) del proyecto, como son el cálculo del mapa de disparidad y la creación de nube de puntos.
- Realizar una comparativa entre los distintos algoritmos de visión estéreo disponibles en la librería de código abierto OpenCV, y poder elegir el más adecuado para su uso en la Jetson TK1.
- Creación del código encargado del cálculo del mapa de disparidades de una imagen y creación de su nube de puntos correspondiente, ambos calculados sobre la GPU y basados en CUDA.
- Creación (en ROS) de los nodos necesarios. Un nodo se encarga del cálculo del mapa de disparidad y otro de la creación de la nube de puntos.

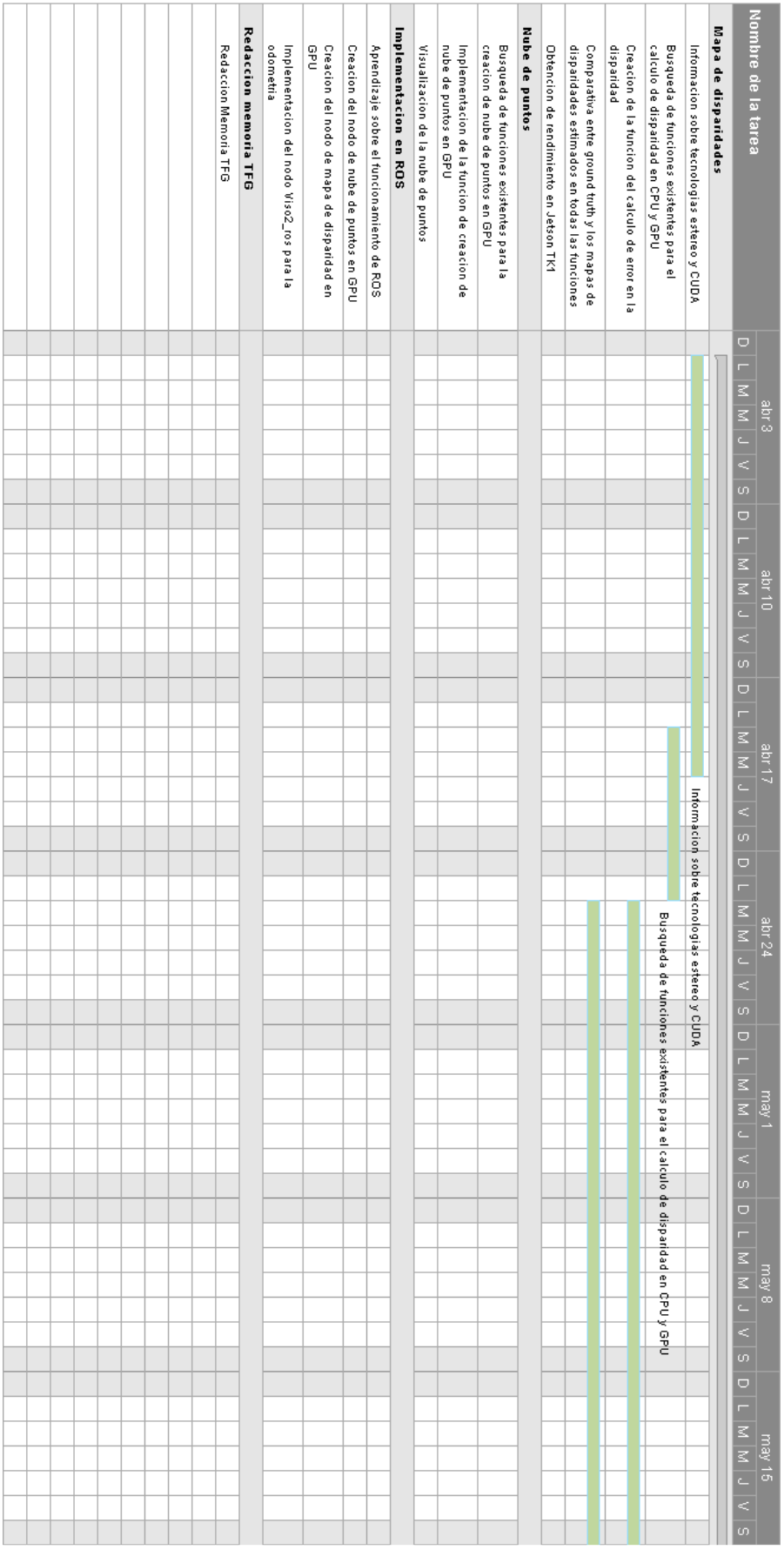
- Integración (en ROS) de los distintos elementos del sistema, como cálculo de odometría, visualización…etc.

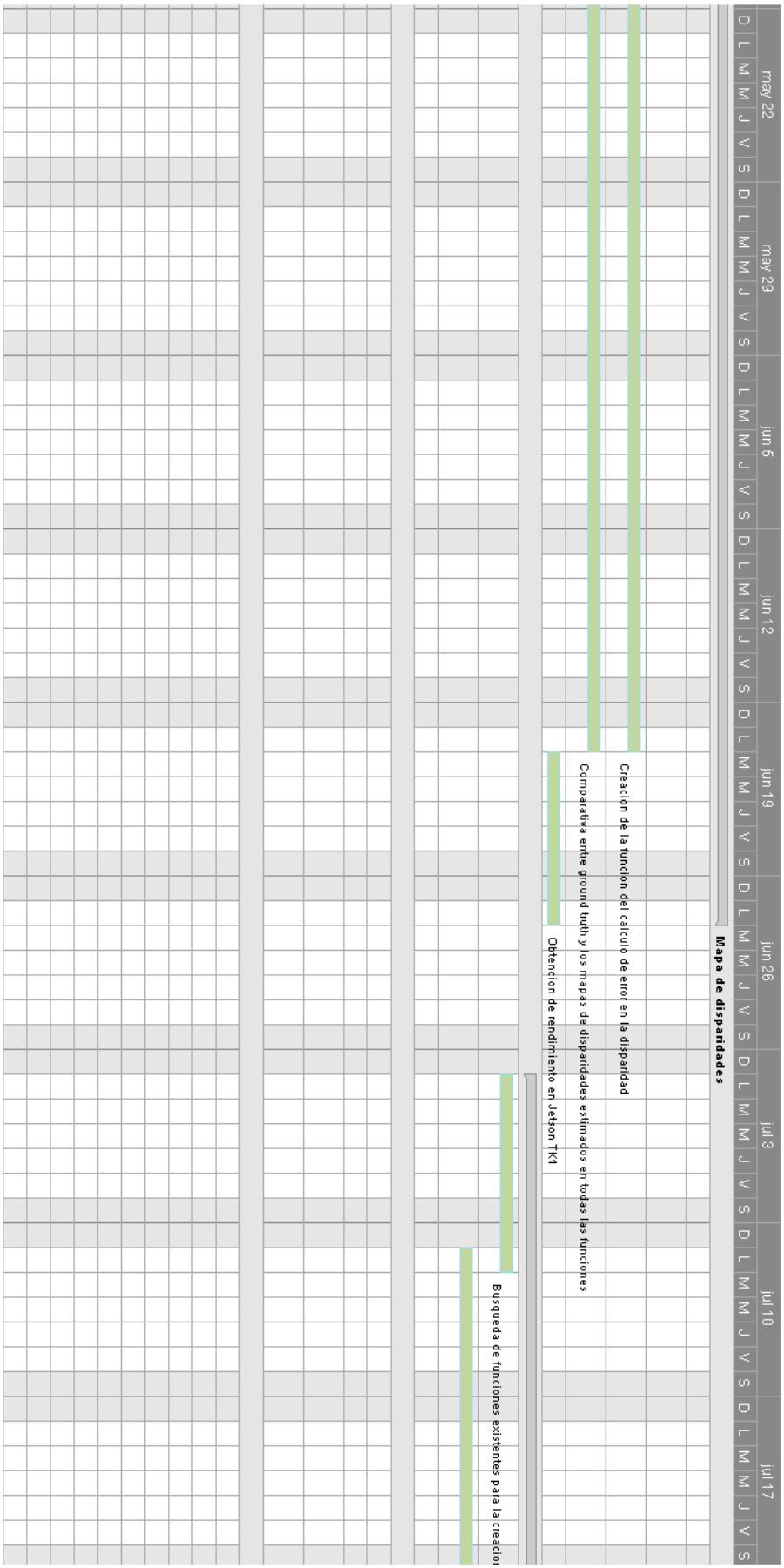
1.3. Planificación

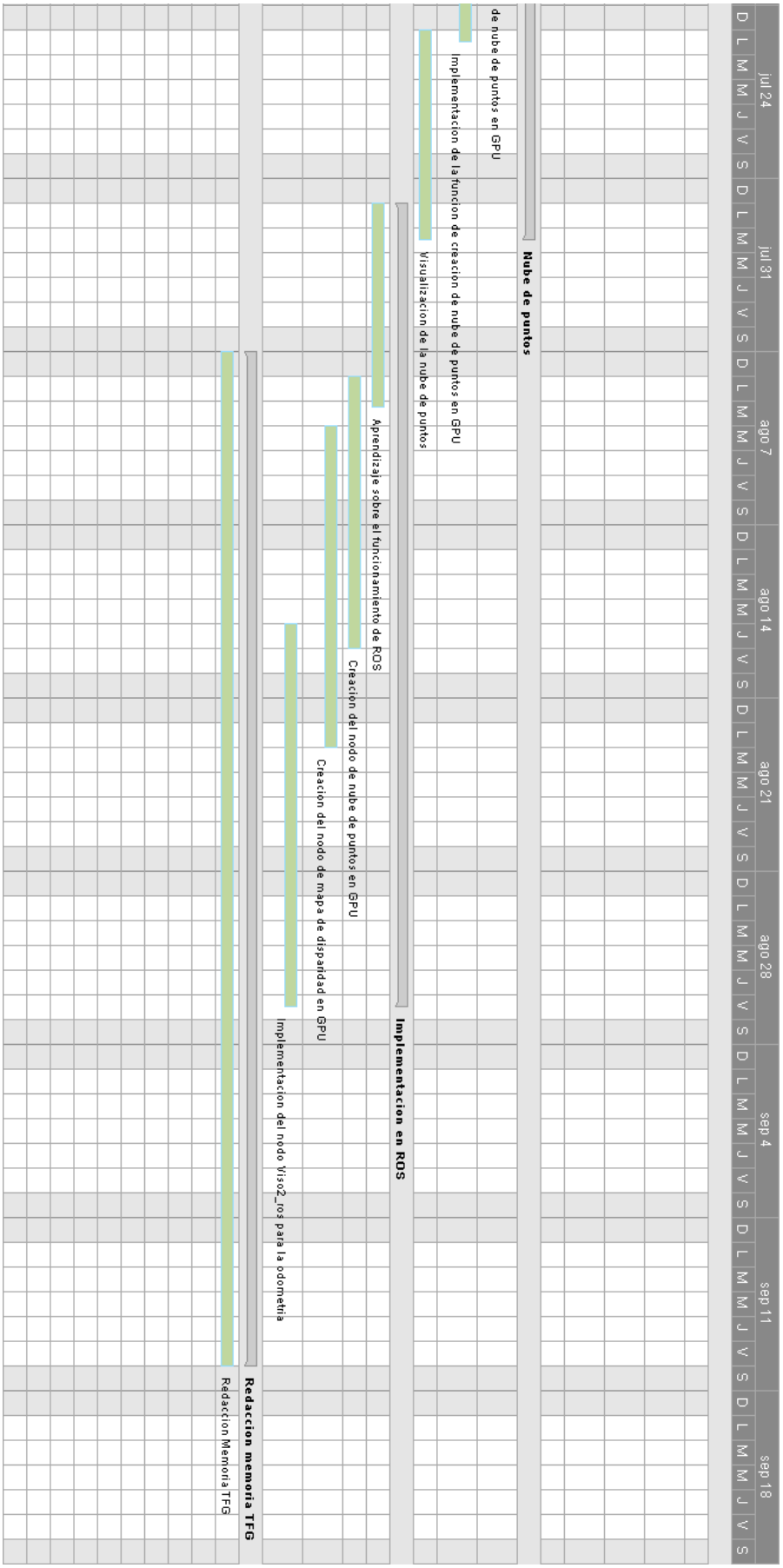
A continuación, se muestra un diagrama de Gantt con una estimación de la duración de este proyecto y sus diferentes etapas. Básicamente se divide en cuatro grandes bloques que corresponden con las distintas etapas que ha llevado el proyecto. En cuanto a la parte técnica, se diferencian tres grandes bloques:

- Mapa de disparidades: Aquí se cuenta el tiempo estimado para la investigación sobre las distintas posibilidades de visión estéreo existentes en las OpenCV, la comparativa entre estos métodos, y finalmente la creación del código final que será implementado en la aplicación.
- Nube de puntos: En este punto también se cuenta el tiempo estimado para la familiarización con las librerías PCL y las distintas posibilidades para la creación de nube de puntos en GPU, así como la creación del código final que será implementado en la aplicación.
- Implementación en ROS: Nuevamente se cuenta el tiempo estimado de adaptación a este entorno de trabajo, y la posterior creación de los nodos necesarios y la integración de todas las partes en la aplicación final.

Finalmente, también se estima el tiempo necesario para la redacción de esta memoria.


Nombre de la tarea
abr 3
abr 10
abr 17
abr 24
may 1
may 8
may 15
Información sobre tecnologías estereo y CUDA
Busqueda de funciones existentes para el calculo de disparidad en CPU y GPU


may 22
may 29
jun 5
jun 12
jun 19
jun 26
jul 3
jul 10
jul 17
Mapa de disparidades
Creacion de la funcion del calculo de error en la disparidad
Comparativa entre ground truth y los mapas de disparidades estimados en todas las funciones
Obtencion de rendimiento en Jetson TK1
Busqueda de funciones existentes para la creación


jul 24
jul 31
ago 7
ago 14
ago 21
ago 28
sep 4
sep 11
sep 18
Nube de puntos
de nube de puntos en GPU
Implementacion de la funcion de creacion de nube de puntos en GPU
Visualizacion de la nube de puntos
Implementacion en ROS
Aprendizaje sobre el funcionamiento de ROS
Creacion del nodo de nube de puntos en GPU
Creacion del nodo de mapa de disparidad en GPU
Implementacion del nodo viso2_ros para la odometria
Redaccion memoria TFG
Redaccion Memoria TFG

1.4. Estructura

Esta memoria se divide en diferentes apartados o secciones, los cuales se enfocan en diferentes etapas o fases del proyecto. Se cree que esta división es las más efectiva para una fácil compresión del trabajo aquí realizado.

a) Se comienza esta memoria por una sección de fundamentos teóricos en los cuales se realiza un repaso sobre los distintos temas que se creen necesarios para la compresión del trabajo realizado. Estos temas abarcan la visión por computador, específicamente la visión estéreo, y las bases de la programación en paralelo basado en CUDA.
b) A continuación, se expone una descripción general del sistema y de todos los componentes que se han utilizado en la realización de este proyecto.
c) El siguiente apartado se centra en la estructura de la aplicación que se pretende desarrollar, basada en ROS y destinada a su funcionamiento en situaciones reales.
d) En los siguientes tres apartados, se describen las distintas etapas del proyecto (Construcción del mapa de disparidad, creación de la nube de puntos, odometría visual e integración en ROS), para acabar con un apartado de resultados, donde se exponen los cálculos realizados y resultados finales.
e) Por último, se exponen las conclusiones alcanzadas tras el proyecto y posibles futuras mejoras que se podrían realizar.

2. Estado del arte

Hoy en día, una gran cantidad de recursos económicos y humanos se están empleando en el desarrollo de tecnologías de reconstrucción 3D, y optimización de aplicaciones con uso de GPGPU. A continuación, se exponen algunas de estas aplicaciones, cuya finalidad varía enormemente de unas a otras, como coches de conducción autónoma o navegación de robots en espacios cerrados, control autónomo de UAV etc.

2.1. Sistemas estéreo

Los sistemas de visión en estéreo se están haciendo cada vez más populares. La existencia de librerías de código abierto como OpenCV junto con el abaratamiento de los costes debido al incremento en la producción de estos sensores están generando que sean la primera elección cuando se quiere dar percepción visual a un sistema robótico inteligente. En el siguiente ejemplo se enseña un ejemplo del uso de estos sensores.

2.1.1. Barrido estéreo para navegación de alta velocidad en espacios abarrotados

En el trabajo realizado por Andrew J. Barry y Russ Tedrake [9], se presenta una técnica de visión estéreo capaz de detectar obstáculos en un procesador móvil ARM a 120 fotogramas por segundo. En este sistema, se realizan una serie de procesamientos de *block-matching,* buscando obstáculos únicamente a una profundidad predeterminada (Figura 9). Se recupera la posición de obstáculos al resto de profundidades usando una unidad de medida inercial *IMU* (*inertial-measurement unit*) y un estimador de estados, construyendo y actualizando un mapa de profundidad *depth-map* a la frecuencia de fotograma.

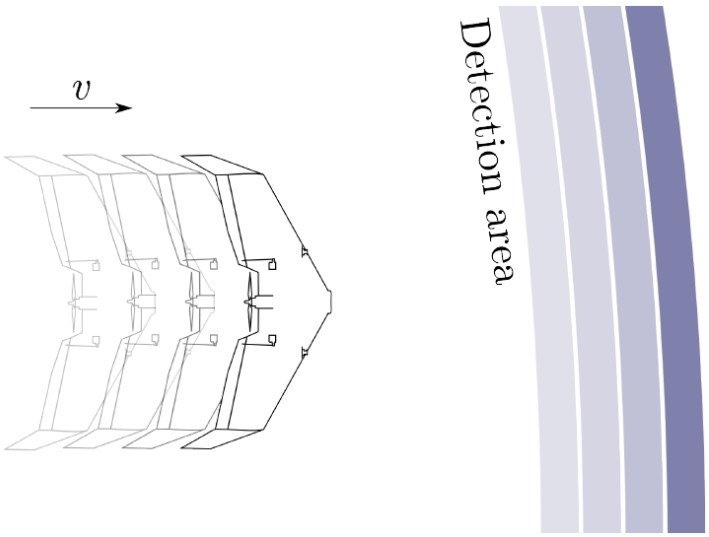


Figura 9: Detectando solo a una profundidad predeterminada (azul oscuro), e integrando la odometría del dron y detecciones anteriores (azul claro), consiguen construir un mapa completo de obstáculos enfrente del vehículo. [9]

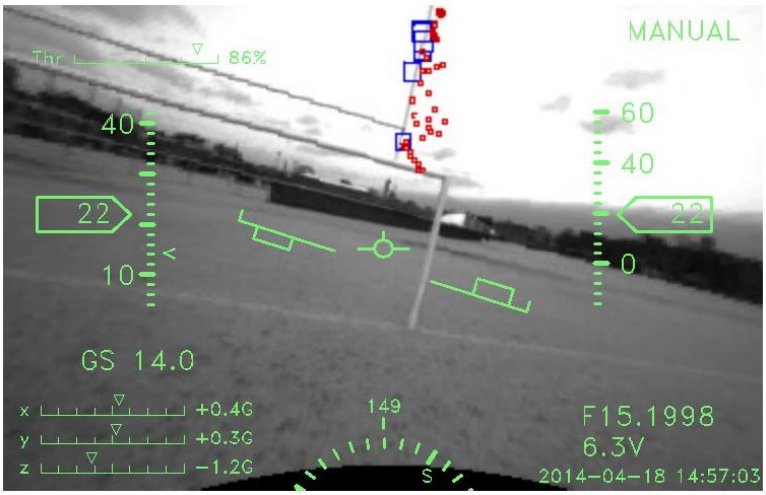


Figura 10: Captura en vuelo de detecciones estéreo en un poste de portería (cajas azules) y pasadas detecciones integradas con la estimación del estado y re-proyectadas en la imagen (puntos rojos). Superpuesto esta información relevante como velocidad en MPH (izquierda) y altitud (derecha). [9]

Se implementa este algoritmo en un pequeño UAV, llegando a velocidades superiores a 20MPH (9 m/s) en vuelos cercanos a obstáculos. En la Figura 10 se muestra una imagen obtenida en uno de estos vuelos autónomos. El sistema no tiene la necesidad de sensores externos o computación externa y proclaman que es el primer sistema de detección estéreo de alta frecuencia de fotogramas, ejecutándose a bordo de un pequeño UAV.

2.2.Sistemas de reconstrucción 3D

Los mapas tridimensionales están resultando ser cada vez más importantes debido a la tendencia actual de diseñar vehículos cada vez más autónomos y al aumento de la cantidad de sistemas robóticos existentes. Cualquier organismo o sistema que pretenda desplazarse de forma autónoma e inteligente por el mundo necesita contar con algún tipo de percepción sensorial para percibir el entorno, pero además debe tener conciencia de su localización o situación en dicho entorno. La generación de mapas en 3D se convierte en la forma más eficiente de obtener que un sistema sea capaz de ser consciente de su posición en el espacio. A continuación, se muestran algunos ejemplos de la creación de estos mapas y su importancia.

2.2.1. Más allá del GPS: cómo mapas en HD mostrarán el camino a los coches autónomos

Los conductores siempre han necesitado mapas, y los coches autónomos no son una excepción. De hecho, la creación de mapas de alta precisión es crucial para convertir los coches autónomos en una realidad.

Este sistema [10] está diseñado para ayudar a fabricantes de coches, compañías de mapas y startups a crear mapas HD y mantenerlos al día, usando la potencia de NVIDIA DRIVE PX 2 en el coche y NVIDIA Tesla GPUs en el centro de datos. Usando 4 cámaras, su plataforma es capaz de detectar hasta 1.8 millones de puntos por segundo en un espacio tridimensional a color, otorgando a los vehículos autónomos una vista completa de sus alrededores. En la Figura 11 se expone una representación de la información adicional que proporcionarían estos nuevos mapas en las carreteras (número de carriles, sentido de circulación, zonas peligrosas etc.).

Los fabricantes de coches tendrán que equipar sus vehículos con potentes ordenadores de abordo capaces de procesar señales de múltiples sensores para poder entender sus alrededores con precisión. Cuando un humano sabe que esperar en la siguiente curva, libera parte de su atención para estar más atento a posibles amenazas. En un coche autónomo no debe ser diferente.

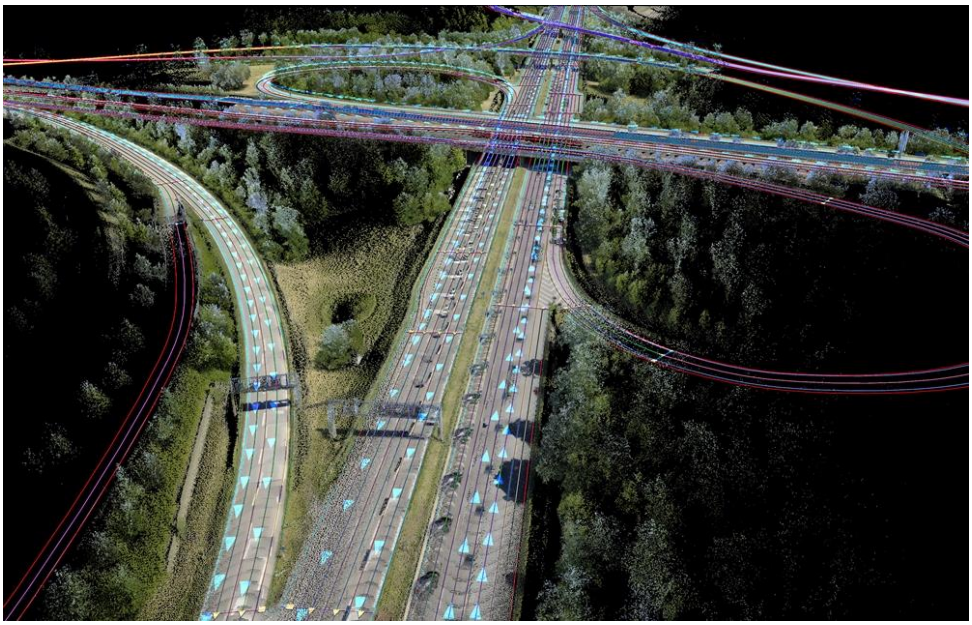


Figura 11: Representación de un posible escenario con información precisa del mapa, como número de carriles, dirección de circulación, salidas… etc. [10]

La creación de información tridimensional, en combinación con información recopilada de sensores inerciales en el coche, y con GPS, permite la localización precisa de características importantes del terreno.

2.2.2. Mapeado RGB-D: mapas 3D densos en interior

Las cámaras RGB-D (RedGreenBlue-Depth) son nuevos sistemas que capturan imágenes RGB junto con información de profundidad por pixel. En la investigación realizada por Peter Henry, Michael Krainin, Evan Herbst, Xiaofeng Ren, Dieter Fox [11], se estudia el uso de este tipo de cámaras en el contexto de la robótica, específicamente en la reconstrucción de mapas 3D densos en entornos cerrados, en donde es más difícil extraer esta información debido a la existencia de zonas sin textura y zonas oscuras. Estos mapas tienen aplicaciones en la navegación, manipulación, mapeado, y tele-presencia robótica.

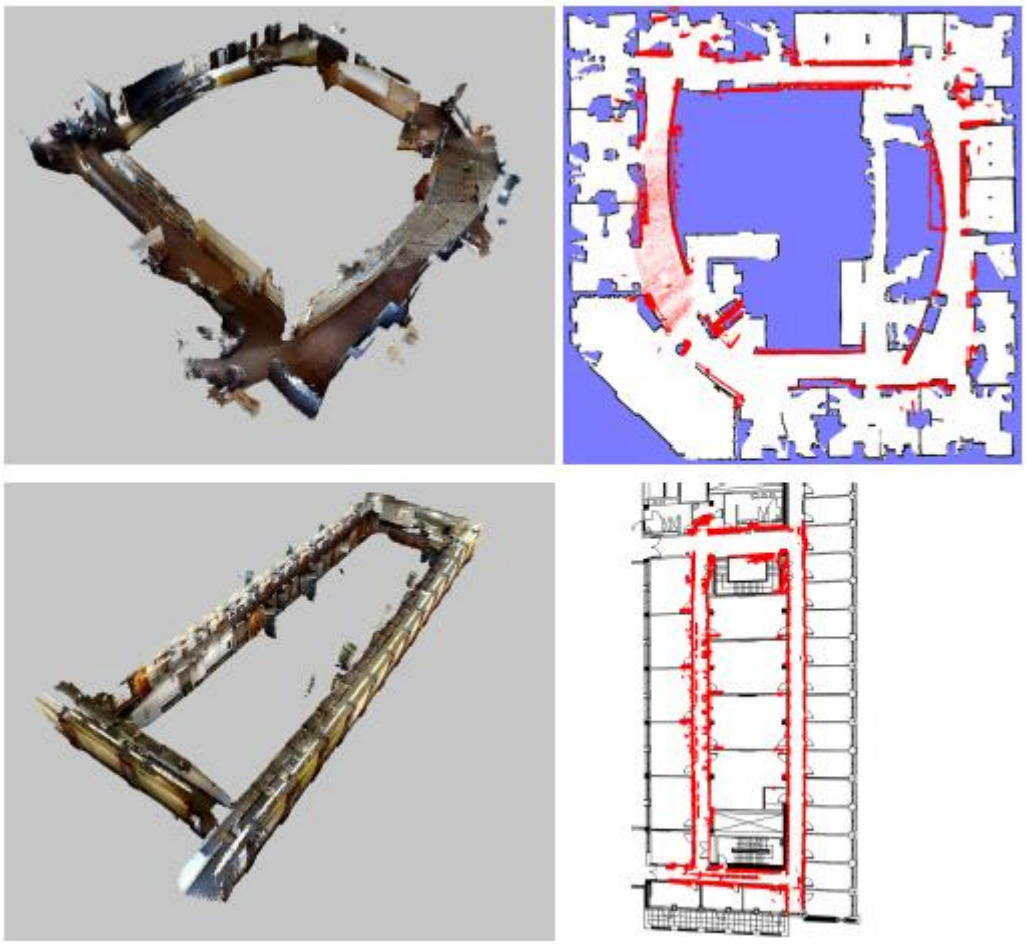
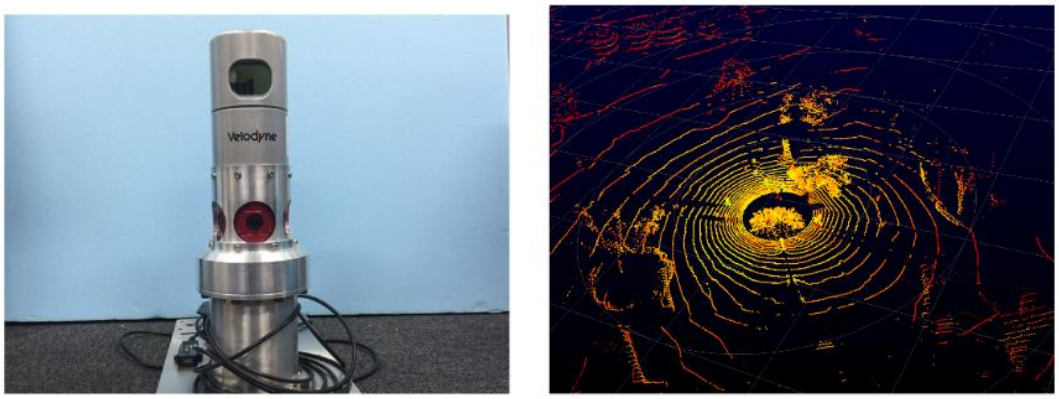


Figura 12: (izquierda) mapas 3D generados con "RGB-D mapping for large loops" en el laboratorio Intel (arriba) y el Allen Center (abajo); (derecha) mapas (rojo) superpuesto en un mapa de escaneado laser del laboratorio Intel y Allen Center. [11]

Información visual y de profundidad son combinadas para una detección de cercanía en bucle basada en imágenes, seguida por una optimización de la pose para obtener mapas consistentes globalmente. Finalmente se obtiene una reconstrucción como la que se puede observar en la Figura 12.

2.2.3. Reconstrucción superficial con sensores Velodyne®

Los sensores de visión en estéreo no es la única opción en tecnologías de reconstrucción tridimensional. Otro tipo de sensor muy utilizado son los escáneres laser, los cuales a pesar de ser bastante más precisos que las tecnologías de cámaras estéreo, presentan un mayor reto en cuanto a velocidad de procesamiento debido al gran coste computacional que conlleva crear las nubes de puntos.



(a) Velodyne® HDL-32E scanner (b) Point Clouds

Figura 13: (a) Sensor velodyne HDL (b) nube de puntos obtenida del sensor [12]

La rápida reconstrucción de superficies con nubes de puntos densas (PCD) es un tema muy recurrente en robótica. Sin embargo, sigue siendo un desafío conseguir mapas de superficie a gran escala con nubes de puntos dispersas (sparse PCD). La investigación realizada por Tianwei Zhang y Yoshihiko Nakamura [12] propone un método para construir mapas de superficie usando la nube de puntos generada por el escáner laser Velodyne® HDL 32E, usando ampliamente en vehículos autónomos y robots de exploración. En esta aplicación se utiliza el método de *supervoxel clustering* para organizar las nubes de puntos y son generadas mallas triangulares representando la escena.

En la Figura 13 se muestra el escáner laser utilizado y la nube de puntos obtenida de éste, mientras que en la Figura 14 y Figura 15 se representan el mapa de normales y los supervoxels de una escena.

Los métodos de creación de nubes de puntos con escáneres láser tienen problemas con la velocidad de procesamiento debido a la gran cantidad de información y problemas de precisión debido al decremento de la densidad de información con la distancia (aun así, mucho más precisos que los métodos de obtención de profundidad con cámaras estéreo). El procesamiento en tiempo real de este tipo de sensores presenta dos cuellos de botella.

Por un lado, conseguir suficiente información estructural espacial del entorno. El escáner Velodyne® HDL-32E rota 32 rayos laser para conseguir una vista 360º. Estos rayos discretos generan como resultado nubes de puntos muy dispersas en distancias lejanas.

Por otro lado, aunque los escáneres láseres han reducido la densidad PCD considerablemente, el sensor Velodyne® HDL-32E genera la enorme cantidad de 70.000 puntos por frame, en unos 2MB en formato pcap. Como es evidente, esto genera una carga computacional enorme.

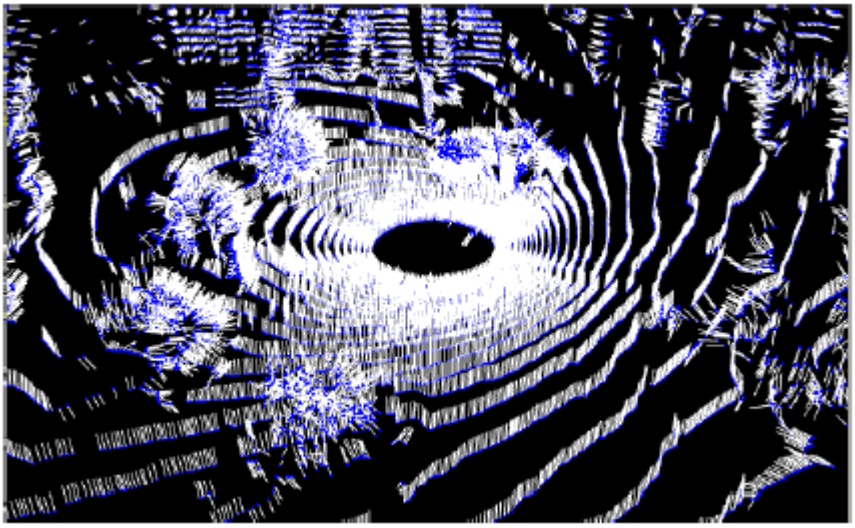


Figura 14: Nube de normales [12]

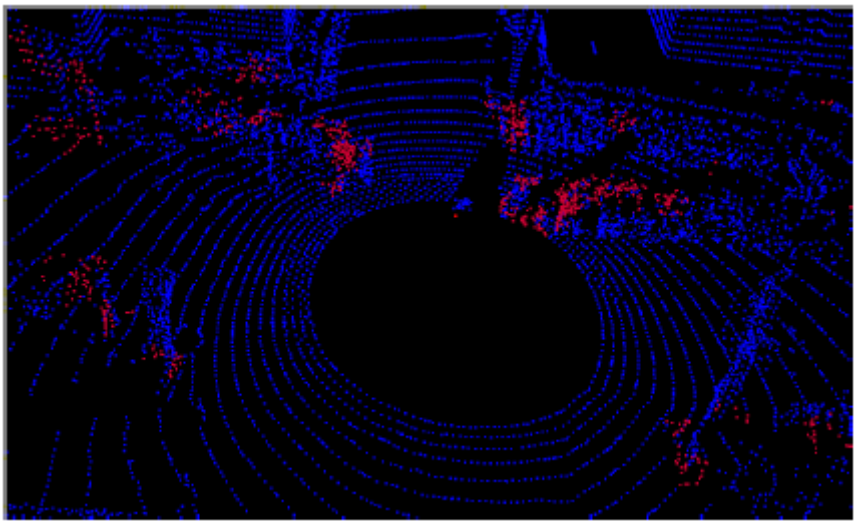


Figura 15: Supervoxels en rojo [12]

2.3.Sistemas GPGPU

En la última década, se ha popularizado el uso de sistemas GPU, originalmente diseñadas para la generación de gráficos, en todo tipo de aplicaciones de carácter científico anteriormente destinadas a ejecutarse en CPU. La ejecución en GPU de estas aplicaciones es realmente efectiva en ámbitos en donde una parte importante del cálculo realizado no es secuencial, y el procesamiento de grandes bloques de datos se puede realizar en paralelo.

2.3.1. Visualización avanzada en tiempo real para cirugía de corazón robótica

Investigadores en el laboratorio de bio-robótica de Harvard están utilizando el poder de computación de las GPU para generar renderizaciones volumétricas del corazón de los pacientes en tiempo real [13]. Han conseguido construir un sistema robótico capaz de manejar de forma autónoma los diversos catéteres comerciales disponibles para adquirir imágenes de ultrasonidos dentro del corazón.

El equipo usó un catéter eco-cardiográfico intracardíaco ICE (Intracardiac Echocardiography), el cual está equipado en el extremo con un transductor de ultrasonidos, para adquirir imágenes 2D del interior del corazón. Usando GPU de NVIDIA, el equipo fue capaz de reconstruir un modelo 4D (3D + tiempo) del corazón con estas imágenes de ultrasonidos. Se puede observar un ejemplo de esta reconstrucción en la Figura 16.

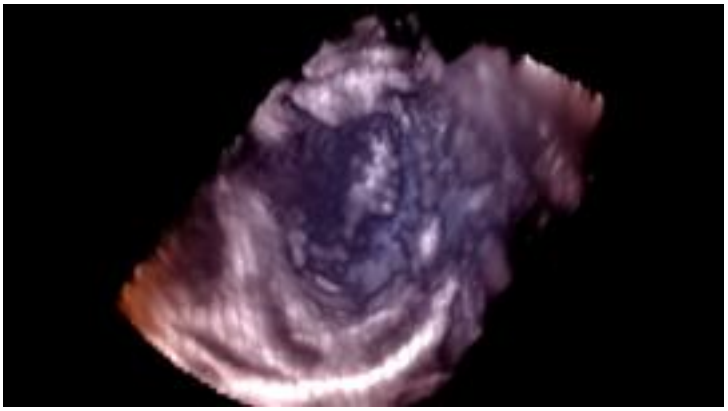


Figura 16: Reconstrucción 3D del interior de un corazón calculada en GPU a base de imágenes de ultrasonidos. [14]

La realidad actual de los catéteres ICE es que son poco frecuentes debido a la dificultad de manejarlos manualmente. Este sistema robótico libera a los clínicos de esta carga, y les presenta un nuevo método de visualización en tiempo real más seguro y de mejor calidad que las imágenes de rayos-X utilizadas en la clínica.

3. Fundamentos teóricos

En esta sección se repasarán ciertos conocimientos teóricos que se consideran necesarios para la completa comprensión de este proyecto. Se abarcarán los temas del modelo matemático para la transformación del espacio tridimensional en una imagen bidimensional, conceptos básicos de la visión estéreo, y finalmente, las bases de funcionamiento de la programación GPGPU en CUDA.

3.1. Modelo Pin-hole

Todas las funciones que se van a utilizar en este proyecto hacen uso del modelo de cámara *pinhole* [15].

Este modelo describe la relación matemática entre las coordenadas de un punto 3D en el espacio, y su proyección en el plano de la imagen de una camera *pinhole* ideal, en donde la apertura de la cámara esta descrita como un punto, y no son utilizadas lentes para enfocar la luz. Una representación de este modelo puede ser la mostrada en la Figura 17.

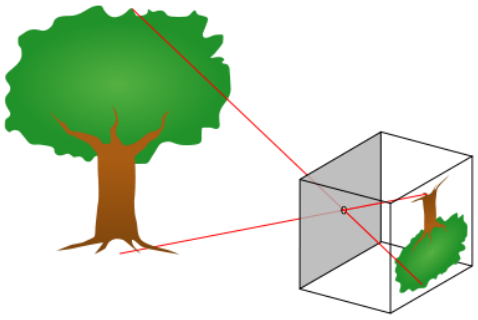


Figura 17: Modelo de cámara pinhole

Con una transformada de la perspectiva:

$$s\, m' = A[R|t]M'$$

o:

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Ecuación 1

En donde:

- (**X, Y, Z**) son las coordenadas de un punto 3D en el espacio de coordenadas exterior
- (**u,v**) son las coordenadas en unidades de pixel del punto proyectado

- **A** es la matriz de los parámetros intrínsecos de la cámara
- (**cx, cy**) es un punto principal que usualmente se encuentra en el centro de la imagen
- **s** es un factor de escala, utilizado en estéreo
- **fx**, **fy** son las distancias focales en unidades de pixel

La matriz de los parámetros intrínsecos de la cámara no depende de la escena observada. Por ello, una vez estimados, pueden ser reutilizados siempre que la distancia focal se mantenga, cosa que no ocurre en lentes con zoom.

La matriz de rotación-traslación $[R|t]$ es llamada matriz de parámetros extrínsecos. Se usa para describir el movimiento de la cámara en torno a una escena estática, o viceversa, el movimiento de un objeto en frente de una cámara estática. La matriz $[R|t]$ traslada las coordenadas de un punto (x,y,z) a un sistema de coordenadas fijado con respecto a la cámara. Esta transformación es equivalente a (cuando z ≠ 0):

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$

$$x' = x/z$$

$$y' = y/z$$

$$u = f_x * x' + c_x$$

$$v = f_y * y' + c_y$$

Las lentes reales suelen tener algún tipo de distorsión, normalmente radial y ligeramente tangencial. Por consiguiente, el modelo anterior quedaría de la siguiente manera:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$

$$x' = x/z$$

$$y' = y/z$$

$$x'' = x' \frac{1+k_1r^2+k_2r^4+k_3r^6}{1+k_4r^2+k_5r^4+k_6r^6} + 2p_1x'y' + p_2(r^2+2x'^2) + s_1r^2 + s_2r^4$$

$$y'' = y' \frac{1+k_1r^2+k_2r^4+k_3r^6}{1+k_4r^2+k_5r^4+k_6r^6} + p_1(r^2+2y'^2) + 2p_2x'y' + s_3r^2 + s_4r^4$$

$$\text{where} \quad r^2 = x'^2 + y'^2$$

$$u = f_x * x'' + c_x$$

$$v = f_y * y'' + c_y$$

Donde k1, k2, k3, k4, k5, y k6 son coeficientes de distorsión radial, p1 y p2 coeficientes de distorsión tangencial, y s1, s2, s3, y s4 coeficientes de distorsión del prisma. Generalmente no se consideran coeficientes de órdenes superiores.

3.2. Visión estéreo

La correspondencia en estéreo o "*Stereo matching*" es el proceso de obtener un modelo 3D de la escena a partir de dos o más imágenes, encontrando la correspondencia entre pixeles en las imágenes, y convirtiendo su posición 2D en una profundidad en 3D.

Los seres humanos percibimos la profundidad gracias a las sutiles diferencias entre las imágenes captadas por el ojo izquierdo y derecho. La tecnología de visión estéreo sigue este principio. Bajo simples configuraciones (ambas cámaras apuntando hacia la misma dirección), la cantidad de movimiento horizontal (diferencia en la localización del objeto en imágenes izquierda y derecha) o *disparidad* es inversamente proporcional a la distancia del objeto al observador.

3.2.1. Geometría epipolar

La Figura 18 muestra como un pixel de una imagen $\mathbf{x_0}$ se proyecta en un *segmento epipolar* en la otra imagen. El segmento está limitado, por un lado, por la proyección en el infinito del rayo de visión original $\mathbf{p_\infty}$, y por el otro lado, por la proyección del centro de la cámara original $\mathbf{c_0}$ en la segunda cámara, el cual es conocida como *epipolo* $\mathbf{e_1}$. Si proyectamos el segmento epipolar de la segunda imagen en la primera imagen, se obtiene otra línea epipolar, limitada esta vez por $\mathbf{e_0}$. Extendiendo estos segmentos hasta el infinito, se obtiene un par de *líneas epipolares*, las cuales son la intersección entre los dos planos de imagen y el *plano epipolar* que pasa por ambos centros de cámara $\mathbf{c_0}$ y $\mathbf{c_1}$ como también por el punto de interés **p**. [16]

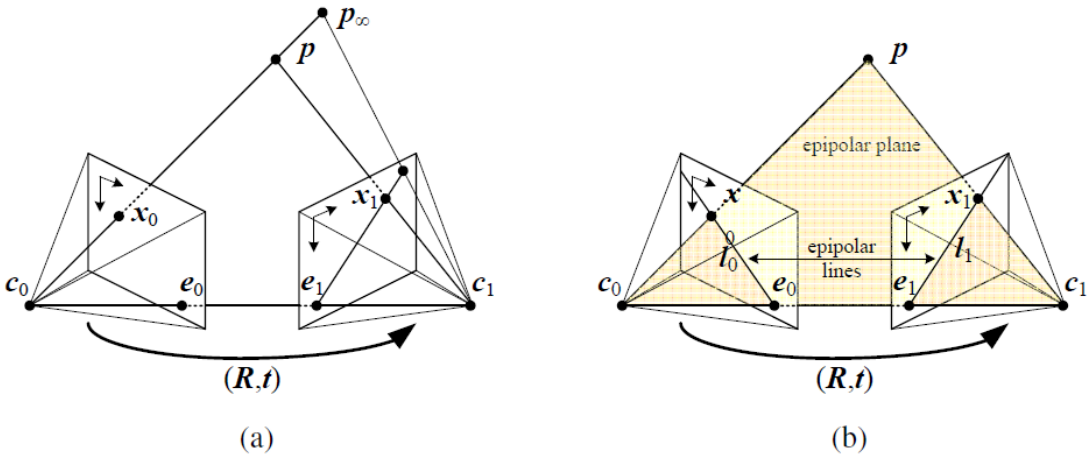


Figura 18: (a) segmento epipolar correspondiente a un rayo; (b) correspondiente par de líneas epipolares y su plano epipolar [16]

3.2.2. Rectificación

Una vez calculada la geometría epipolar, se puede usar la línea epipolar correspondiente a un pixel en una imagen, para limitar la búsqueda del pixel correspondiente en la otra imagen.

Una manera de abordarlo es usando un algoritmo de correspondencia general, como puede ser optical flow [17], considerando solo pixeles situados en la línea epipolar.

Se puede conseguir un algoritmo más eficiente *rectificando* en primer lugar las imágenes originales de tal forma que las líneas epipolares se conviertan en líneas horizontales correspondientes [18]. Después, es posible emparejar líneas horizontales independientemente, o desplazar imágenes horizontalmente mientras se calcula las puntuaciones de emparejamiento.

Una manera sencilla de rectificar ambas imágenes consiste en primero, rotar ambas cámaras para que estén mirando perpendicularmente a la línea de unión entre ambos centros de cámara $\mathbf{c_0}$ $\mathbf{c_1}$. Ya que hay un pequeño grado de libertad en la inclinación, se debe usar las rotaciones mínimas que cumplan este objetivo. A continuación, para determinar la rotación deseada alrededor del eje óptico, hacer que el vector vertical (el eje y de la cámara) sea perpendicular a la línea del centro de cámara. Esto garantiza que las líneas epipolares correspondientes sean horizontales y la disparidad de puntos en el infinito sea 0. Finalmente, si es necesario, se re-escalan las imágenes para contrarrestar diferentes distancias focales, aumentando la imagen más pequeña para evitar el efecto de *aliasing*. Las imágenes resultantes después de cada operación se muestran en la Figura 19.

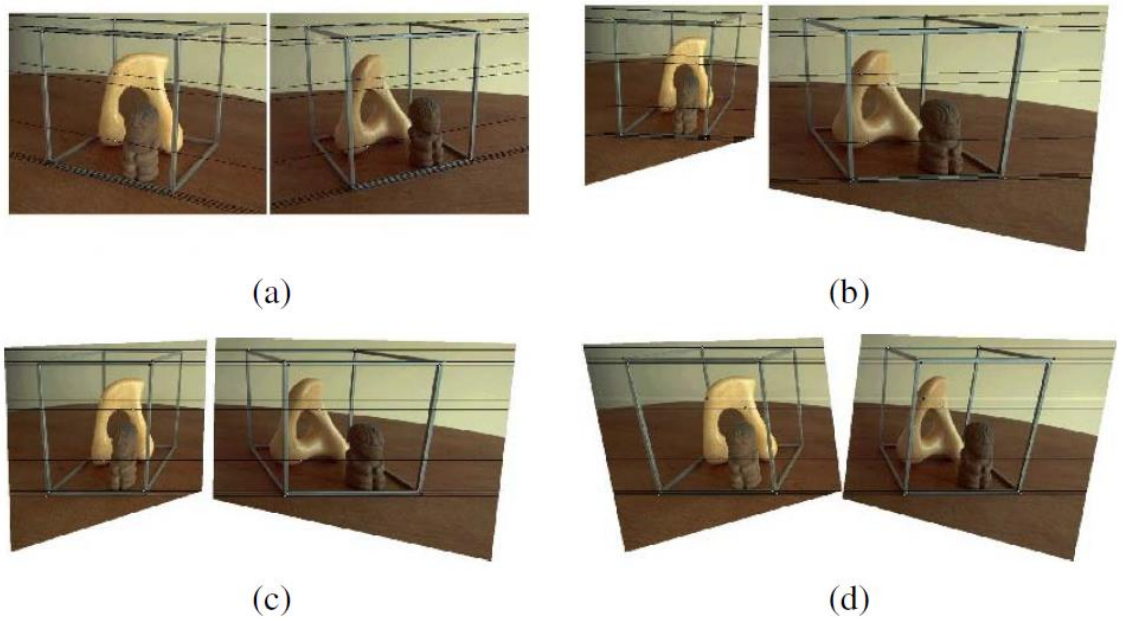


Figura 19: El algoritmo multi-etapa de rectificación estéreo de Loop y Zhang [18]. (a) Imagen original con varias líneas epipolares superpuestas; (b) imágenes transformadas para que las líneas epipolares sean paralelas; (c) imágenes rectificadas para que las líneas epipolares sean horizontales y en correspondencia vertical; (d) rectificación final que minimiza distorsiones horizontales.

La geometría rectificada estándar resultante es usada en numerosas configuraciones de cámaras estéreo y algoritmos estéreo, y conlleva una simple relación inversa entre profundidades 3D Z y disparidades d:

$$d = f\frac{B}{Z}$$

En donde f es la distancia focal (medida en pixeles), B es el *baseline* (la distancia variable entre cámaras) y donde:

$$x' = x + d(x, y)$$

$$y' = y$$

Describe la relación existente entre coordenadas de pixeles correspondientes en la imagen izquierda y derecha.

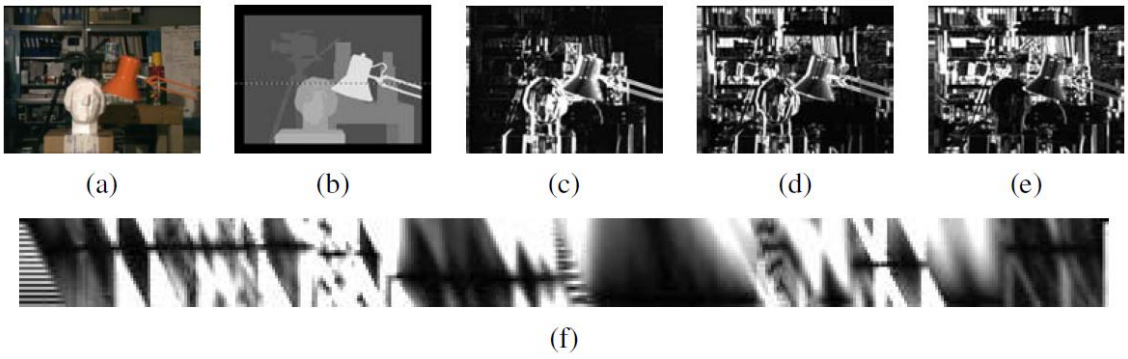


Figura 20: Secciones de un típico disparity space image (DSI) (Scharstein and Szeliski 2002) © 2002 Springer: (a) imagen original en color; (b) disparidades del ground truth; (c-e) tres seciones o rebanadas (x,y) para d=10,16,21; (f) una sección o rebanada (x,d) para y= 151 (la línea punteada en (b)).Varias regiones oscuras son visibles en (c-e). Por ejemplo, las estanterías, mesa y latas, y la estatua se pueden ver como tres líneas horizontales. Las bandas oscuras en el DSI indican regiones que corresponden a dicha disparidad

La tarea de extraer la profundidad a partir de un conjunto de imágenes se convierte en la de estimar el *mapa de disparidad* $d(x, y)$. Despues de la rectificación, se puede comparar fácilmente la similaridad de pixeles en localizaciones correspondientes (x, y) y $(x', y') = (x + d, y)$ y guardarlos en una imagen de espacio de disparidades o *disparity space image* (DSI) $C(x, y, d)$ para procesamientos posteriores. En la Figura 20 se observan distintas secciones del mapa de disparidad a diferentes profundidades y el espacio de disparidades o (DSI) $C(x, y, d)$ para una sección de la imagen (x,d) con y=151, de una imagen dada (a). Por ejemplo, las estanterías, mesa y latas, y la estatua se pueden ver como tres líneas horizontales. Las bandas oscuras en el DSI indican regiones que corresponden a dicha disparidad.

3.2.3. Correspondencia densa

La mayoría de los algoritmos de correspondencia estéreo actualmente se centran en la correspondencia densa, ya que este es un requerimiento para aplicaciones como modelado o renderización basado en imágenes. Esto presenta un gran reto, ya que estimar valores de profundidad de regiones sin textura requiere de una cierta cantidad de suposiciones o conjeturas.

En el libro publicado por Szeliski [16], se revisa un esquema de clasificación de algoritmos de correspondencia densa propuestos por Scharstein and Szeliski (2002). La

clasificación consiste en un conjunto de bloques constructivos o "building blocks", a partir de los cuales se construye un mayor conjunto de algoritmos. Se basa en la observación de que los algoritmos de visión estéreo generalmente desarrollan algunos de los siguientes 4 pasos:

I. Cálculo del coste de emparejamiento (matching cost computation)
II. Acumulación del coste -coste agregado/acumulado- (cost aggregation)
III. Calculo de disparidad y optimización
IV. Refinamiento de la disparidad

Por ejemplo, los métodos locales (basados en ventanas, como puede ser el algoritmo BlockMatching (BM) que se verá posteriormente), donde el cálculo de disparidad en un cierto punto depende únicamente de la intensidad dentro de una ventana finita, normalmente hacen asunciones de uniformidad implícitas sumando el coste. Algunos de estos algoritmos pueden ser claramente separados en 3 pasos. Por ejemplo, el tradicional algoritmo suma de diferencia de cuadrados o "sum-of-squared-differences" (SSD) puede ser descrito como:

1. El coste de emparejamiento o "matching cost" es la diferencia de cuadrados de los valores de intensidad a una disparidad dada.
2. La acumulación se realiza sumando el coste de emparejamiento sobre ventanas cuadradas con disparidad constante.
3. Las disparidades son calculadas al seleccionar el mínimo valor del coste acumulado por cada pixel.

El primer componente de cualquier algoritmo de emparejamiento estéreo (dense stereo matching) es una medida de similitud que compare valores de pixel con el objetivo de determinar que probabilidad tienen estos de estar en correspondencia. Entre los métodos más comunes basados en pixel de coste de emparejamiento se incluyen suma de diferencia de cuadrados o "sum-of-squared-differences" (SSD) (Hannah 1974) y suma de diferencias de intensidad absolutas o "sum of absolute intensity differences" (SAD) (Kanade 1994).

Los métodos locales y basados en ventanas acumulan el coste de emparejamiento sumando o promediando sobre una región en el (DSI) $C(x, y, d)$. Esta región puede ser bidimensional a una disparidad determinada fija, o tridimensional en (x, y, d). Ejemplos de métodos locales puede ser el "block matching" (BM), el cual se utilizará en las fases posteriores de este proyecto.

Los algoritmos globales, por el contrario, realizan asunciones de uniformidad explicitas y, después, solucionan un problema de optimización global. Típicamente, estos algoritmos no realizan el paso de la acumulación del coste, sino que directamente buscan asignar una disparidad (paso 3) que minimice una función de coste global, que consta de condiciones de datos (paso 1) y condiciones de uniformidad. La mayor diferencia entre estos algoritmos es el procedimiento de minimización utilizado. Un ejemplo de método global podría ser el BeliefPropagation, el cual también será utilizada en una etapa del proyecto posterior.

3.3. Programación en CUDA

CUDA® es una plataforma de computación en paralelo y una interfaz de programación de aplicación (API - Application Programming Interface) creada por NVIDIA. CUDA permite a desarrolladores de software e ingenieros del software utilizar la tarjeta gráfica habilitada con

CUDA para procesamientos de propósito general (un enfoque conocido como General Purpose Graphics Processing Unit ó **GPGPU**). La plataforma CUDA es una capa de software que da acceso directo al set de instrucciones virtual y a elementos computacionales en paralelo de la GPU, para la ejecución de *kernels*. Un kernel en "C for CUDA", es una función la cual al ejecutarse lo hará en N distintos hilos en lugar de en secuencial.

La plataforma de CUDA está diseñada para trabajar con los lenguajes de programación C/C++ y Fortran. Esta accesibilidad facilita a los programadores la utilización de recursos de la GPU, sin necesidad de tener conocimientos avanzados de programación de gráficos, como era el caso de previas soluciones API como Direct3D y OpenGL.

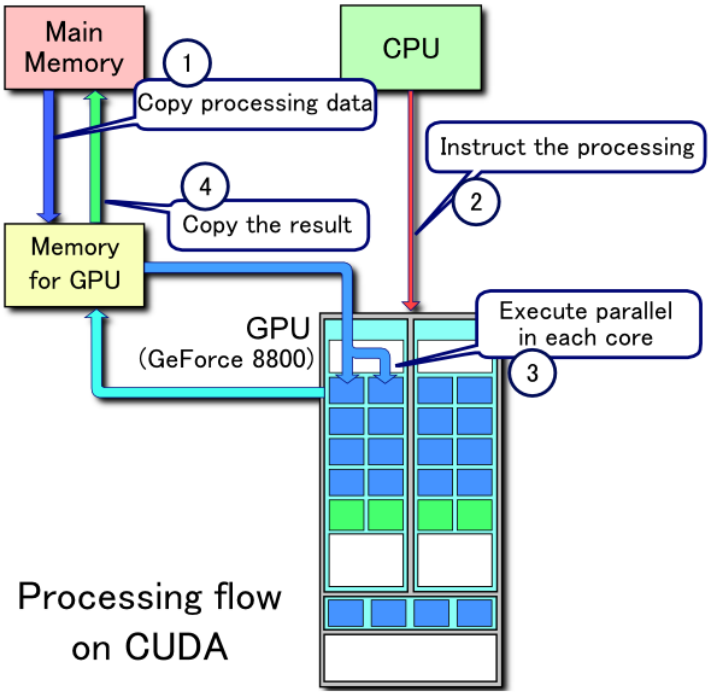


Figura 21: Ejemplo del flujo de procesamiento CUDA [19]

La GPU, siendo un procesador especializado, aborda la demanda de tareas de computación intensiva de gráficos 3D de alta resolución en tiempo real. Por consiguiente, la arquitectura de las GPU ha evolucionado en sistemas multi-núcleo de gran paralelización que permiten una manipulación muy eficiente de grandes bloques de datos. Esta arquitectura es más efectiva que las CPUs de propósito general para algoritmos en donde el procesamiento de grandes bloques de datos es realizado en paralelo.

La estructura en el modelo de programación en CUDA está definida por una retícula (*grid)*, dentro del cual hay bloques de hilos (*threads)* que están formados por 512 hilos distintos como máximo. Cada hilo está identificado con un identificador único, que se accede con la variable *threadIdx*. Esta variable es muy útil para repartir el trabajo entre distintos hilos. La variable *threadIdx* tiene 3 componentes (x, y, z), coincidiendo con las dimensiones de bloques de hilos. Por ejemplo, cada elemento de una matriz lo podría tratar su homólogo en un bloque de hilos de dos dimensiones. En la Figura 21 se muestra una representación del flujo de procesamiento bajo la arquitectura CUDA.

El multiprocesador crea y maneja los hilos sin ningún tipo de *overhead* por la planificación, lo cual unido a una rápida sincronización por barreras y una creación de hilos muy ligera, consigue que se pueda utilizar CUDA en problemas de muy baja granularidad, incluso asignando un hilo a un elemento de una imagen (un píxel).

4. Descripción general del sistema

Este proyecto consta de diversos componentes de hardware, junto con diferentes elementos de software, los cuales pasaran a ser detallados a continuación. También se explica brevemente el objetivo inicial de la aplicación y la integración de estos componentes hardware/software.

4.1.Introducción a la aplicación

Antes de comenzar el desarrollo de este proyecto se realizó una valoración inicial sobre los objetivos a cumplir y las diferentes posibilidades de desarrollarlo.

Una de las ideas previas a la ejecución de este proyecto, fue la de experimentar con una filosofía de computación distribuida. Actualmente, en el vehículo de investigación del LSI-Laboratorio de Sistemas Inteligentes IVVI (Intelligent Vehicle based on Visual Information), todo el procesamiento se realiza en el mismo equipo. La idea inicial fue la de liberar en cierta medida la carga computacional de este potente equipo central, dividiendo las diferentes tareas en módulos más o menos independientes, que se puedan ejecutar en equipos auxiliares.

La reciente adquisición en el departamento del sistema embebido Jetson TK1, proporcionó la herramienta necesaria para cumplir este objetivo. Este sistema se especializa en la utilización de la GPU para algoritmos de propósito general o GPGPU. En ciertas áreas, estos algoritmos tienen un rendimiento notablemente superior a los ejecutados en CPU.

El sistema Jetson TK1 está diseñado específicamente para trabajar con la plataforma de computación en paralelo CUDA de NVIDIA. Trabajando sobre este sistema, se probaron diferentes algoritmos ya existentes en las librerías OpenCV programados específicamente para CUDA, y se compararon con algoritmos en CPU más conocidos. Una vez escogido el método más adecuado en cuanto a tiempo y precisión, se procedió a realizar una reconstrucción en 3D de la escena. Generando nubes de puntos a partir de las imágenes, se consiguió visualizar un mapa en tres dimensiones de la zona grabada.

Finalmente, se obtuvo un sistema compacto y de bajo consumo, en una plataforma embebida con capacidades GPGPU de programación paralela heterogénea. El aumento de la velocidad de procesamiento en ciertos aspectos que ofrecen las GPUs permite cumplir restricciones de funcionamiento en tiempo real consiguiendo un sistema compacto y de bajo consumo.

4.2.Hardware

A continuación, se exponen las características y especificaciones técnicas de los equipos utilizados durante la realización de este proyecto.

4.2.1. Características del microcontrolador Jetson TK1

Figura 22: NVIDIA Jetson TK1

La plataforma NVIDIA Jetson TK1 (Figura 22) [20] es la primera de su clase que cuenta con la misma arquitectura y características que una tarjeta gráfica de escritorio GPU moderna, manteniendo un bajo consumo propio de los procesadores móviles. Esto ofrece al usuario la posibilidad de trabajar con un sistema embebido y usar el mismo código CUDA que se ejecuta en una GPU de escritorio.

La plataforma de desarrollo de NVIDIA Jetson TK1 basada en Linux consta de un Tegra K1 SoC (CPU + GPU + ISP en un solo chip). A continuación, se exponen las características del sistema, que incluye características similares a una Raspberry Pi, pero también características orientadas a PC como podría ser SATA, mini-PCIe (*Peripheral Component Interconnect Express*) o un pequeño ventilador que permite un funcionamiento continuo bajo grandes cargas de trabajo:

- **Dimensiones:** 127mm x 127mm placa
- **Tegra K1 SOC** (CPU+GPU+ISP in a single chip, with typical power consumption between 1 to 5 Watts):
    - **GPU:** NVIDIA Kepler "GK20a" GPU con 192 núcleos SM3.2 CUDA (hasta 326 GFLOPS)
    - **CPU:** NVIDIA "4-Plus-1" 2.32GHz ARM quad-core Cortex-A15 CPU with Cortex-A15 battery-saving shadow-core
- **DRAM:** 2GB DDR3L 933MHz EMC x16 con 64-bit data width
- **Storage:** 16GB fast eMMC 4.51 (routed to SDMMC4)
- **mini-PCIe:** un puerto de baja altura single-lane PEX (para por ejemplo Wifi, SSD RAID, FireWire o tarjetas addon Ethernet)
- **SD/MMC card:** ranura tamaño completo (routed to SDMMC3)
- **USB 3.0:** una ranura hembra Tipo-A. a full-size Type-A female socket
- **USB 2.0:** una ranura hembra tipo micro-AB (para conectar a un PC, pero tambien puede ser usada como un Puerto USB 2.0 adicional usando un adaptador micro-B macho a tipo-A hembra)
- **HDMI:** un puerto tamaño completo
- **RS232:** un puerto DB9 a tamaño completo (routed to UART4)
- **Audio:** un codec de audio ALC5639 Realtek HD con puertos de entrada y salida de audio (routed to DAP2)
- **Ethernet:** un puerto RTL8111GS Realtek 10/100/1000Base-T Gigabit LAN usando PEX

- **SATA:** un puerto de tamaño completo que soporta discos de 2.5” y 3.5” pero no es hot-pluggable. (Apagar antes de conectar algun disco SATA)
- **JTAG:** un puerto 2x10-pin 0.1" para debugging
- **Power:** un puerto barrel-type 12V DC y un PC IDE conector de 4 pines, usando AS3722 PMIC
- **Fan:** un ventilador alimentado por 12V (para permitir el uso intenso del aparato bajo grandes cargas de trabajo, aunque puede ser sustituido por otro disipador de calor)

Las siguientes señales están disponibles a través del puerto de expansión 125-pin 2mm-pitch:

- **Camera ports:** 2 puertos rapidos para cámara CSI-2 MIPI (uno 4-lane y el otro 1-lane)
- **LCD port:** LVDS y eDP Display Panel
- **Touchscreen ports:** Touch SPI 1 x 4-lane + 1 x 1-lane CSI-2
- **UART**
- **HSIC**
- **I2C:** 3 puertos
- **GPIO:** 7 x GPIO pins (1.8V). Los pines de la cámara CSI pueden ser usados como GPIO extras si no se utilizan ambas cámaras

Panel de conexión frontal:

- Verde - Power LED
- Naranja - HDD LED
- Rojo - Power Button
- Morado/Azul - Reset Button

APIs con aceleración de hardware soportadas

- **CUDA** 6.0 (SM3.2, prácticamente el mismo que el de escritorio SM3.5)
- **OpenGL** 4.4
- **OpenGL ES** 3.1
- **OpenMAX IL multimedia codec** incluyendo H.264, VC-1 y VP8 a traves de Gstreamer
- **NPP** (CUDA optimized NVIDIA Performance Primitives)
- **OpenCV4Tegra** (NEON + GLSL + optimizaciones quad-core CPU)
- **VisionWorks**

Gracias al puerto mini-PCIe, se ha podido añadir una tarjeta Firewire a través de la cual se puede conectar la cámara estéreo, de la que se hablará a continuación. En la Figura 23 y Figura 24 se muestra el estado actual de la Jetson TK-1 en el grupo de investigación LSI- Laboratorio de Sistemas Inteligentes, con el puerto Firewire instalado.

Figura 23: Lateral Jetson TK1 en el departamento LSI

Figura 24: Lateral Jetson TK1 en el departamento LSI

4.2.2. Características de la cámara estéreo Pointgrey Bumblebee XB3

Se ha utilizado la cámara estéreo Pointgrey Bumblebee XB3 BBX3-13S2C-60 (Figura 25) para la segunda parte del proyecto, que consiste en la reconstrucción 3D de la escena. Es una cámara estéreo de 3 sensores 1.3 mega-pixeles con dos baselines disponibles para el uso. Uno

extendido para mayor precisión a distancias grandes, y otro reducido, para mejorar las limitaciones en las distancias cortas.

Figura 25: Cámara estéreo Pointgrey Bumblebee XB3 BBX3-13S2C-60

Esta cámara está perfectamente integrada con el meta sistema operativo ROS, utilizado por el departamento. Y sus principales características técnicas son:

- **Resolution**: 1280 x 960
- **Frame Rate:** 16 FPS
- **Megapixels**: 1.3 MP
- **Chroma**: Color
- **Sensor Name**: Sony ICX445
- **Sensor Type**: CCD
- **Readout Method**: Global shutter
- **Sensor Format**: 1/3"
- **Pixel Size**: 3.75 µm
- **Lens Mount**: 3 x M12 microlens
- **Focal Length**: 6 mm, 43-deg HFOV
- **Aperture**: f/2.5
- **ADC**: 12-bit
- **Exposure Range**: 0.03 ms to 66.63 ms
- **Trigger Modes**: Standard, bulb, skip frames, overlapped
- **Flash Memory**: 512 KB non-volatile memory
- **Non-isolated**: I/O Ports 4 bi-directional
- **Serial Port**: 2 RS-232 (dedicated pins)
- **Auxiliary Output**: 3.3 V, 150 mA maximum
- **Interface**: FireWire 1394b
- **Power Requirements**: 12 V
- **Power Consumption** (Maximum): 4 W at 12 V
- **Dimensions**: 277 mm x 37 mm x 41.8 mm
- **Mass**: 505 grams
- **Machine Vision Standard**: IIDC v1.31

- **Compliance**: CE, FCC, KCC, RoHS
- **Temperature** (Operating): 0° to 45°C
- **Temperature** (Storage): -30° to 60°C
- **Humidity** (Operating): 20 to 80% (no condensation)
- **Humidity** (Storage): 20 to 95% (no condensation)

4.2.3. Características de portátil ASUS R510J

Ordenador portátil ASUS R510J utilizado en el proyecto. Su uso fue debido a la existencia de una única placa Jetson TK1 en el LSI, la cual no podía ser extraída de la Universidad por motivos de seguridad. Por ello se utilizó un portátil con una tarjeta gráfica de NVIDIA GeForce 930M para trabajar más cómodamente sin necesidad de trasladarse a la universidad cada vez que se quería avanzar en el proyecto.

Dado que el chip de la Jetson (Tegra K1 SoC) está diseñado bajo la misma arquitectura NVIDIA Kepler™ que un ordenador de sobremesa, no hay ningún problema de compatibilidad y puede ejecutarse prácticamente el mismo código sin ningún problema. Las especificaciones se exponen a continuación:

- Procesador Intel® Core™ i7 (i7-4720HQ) Quad-Core (6M Cache, 2.6GHz hasta 3.6GHz)
- Memoria RAM 4GB (4GB [ON BOARD]) DDR3L 1600MHz
- Disco duro 1TB 5400rpm SATA
- Almacenamiento óptico DVD 8X Supermulti, Doble Capa [SATA]
- Pantalla 15.6" LED Retroiluminado / Ultra Slim 200nits / Full HD (1920x1080/16:9) / Anti-Glare / NTSC:45%
- Controlador gráfico NVIDIA® GeForce® 930M
- Conectividad
  - Red inalámbrica 802.11 bgn
  - Bluetooth 4.0
  - Red 10/100/1000 Mbps
- Cámara de portátil VGA (640x480)
- Micrófono integrado
- Batería 44WHrs, 4 celdas Li-ion
- Conexiones
  - 1 x USB 2.0
  - 2 x USB 3.0
  - 1 x Entrada/Salida línea audio (combo)
  - 1 x Conector RJ45 LAN
  - 1 x VGA (D-Sub)
  - 1 x HDMI
  - 1 x Orificio Bloqueo Kensington
  - 1 x Entrada de Corriente
- Lector de Tarjetas SD(SDHC/SDXC)
- Dimensiones (Ancho x Profundidad x Altura) 380 x 251 x 29,2~31,7 mm
- Peso 2.5Kg

4.2.4. IVVI 2.0

El IVVI 2.0 es una plataforma de investigación del Laboratorio de Sistemas Inteligentes (LSI) [21] de la universidad Carlos III de Madrid, para la implantación de sistemas inteligentes basados en visión por computador y técnicas de láser, con el objetivo de desarrollar y probar tecnologías ADAS (Advanced Driver Assistance Systems). El propósito del IVVI es probar nuevos algoritmos en condiciones reales.

La plataforma IVVI 2.0 cuenta con el más avanzado equipamiento, y ha sido diseñado pensando en las futuras necesidades de los sistemas de conducción asistida. Todos los ordenadores, sensores e interfaz humano-maquina están perfectamente integrados en el vehículo y son prácticamente invisibles. En la Figura 26 se puede observar el vehículo y los diversos sensores que este utiliza, entre los que se puede encontrar:

- Un sistema de visión en estéreo para detección y clasificación de objetos (vehículos, peatones, etc.), carretera y señales de tráfico.
- Una cámara infrarroja (FIR - far infrarred range), montada en el retrovisor, pensada para la detección de peatones en condiciones de conducción nocturna.
- Un escáner laser para detección de objetos montado en el frente del parachoques.
- Un sensor de movimiento para detección de rostros y monitorización de la conducción, situado en el salpicadero.
- Un aparato de comunicación bus CAN, basado en un sistema embebido de bajo consumo, enfocado al análisis del comportamiento del conductor.
- Un receptor GNSS y una unidad de medida inercial (IMU - Inertial Measurement Unit), integrados en el techo del vehículo IVVI, con el objetivo de recopilar información acerca de la posición y movimiento del vehículo.

Además, dos nuevas cámaras enfocadas a los laterales han sido incorporadas en el 2016, con el objetivo de detectar vehículos en rotondas y cruces. Los datos recibidos de los sensores son procesados en una plataforma de computación situada en el maletero, la cual es capaz de trabajar en tiempo real. Con el fin de proporcionar información al conductor está conectado a los altavoces del vehículo y diferentes mensajes de advertencia de audio y alertas son emitidos. Además, una pantalla en el salpicadero del conductor muestra advertencias visuales y alertas de peligro.

Figura 26: Dispositivos a bordo del IVVI [21]

La arquitectura del software está basada en ROS, habilitando la fusión de información de bajo nivel y alto nivel. La ventaja de la sincronización de datos de bajo nivel a través de ROS es la marca de tiempo de adquisición de datos. Por lo tanto, se ha mejora la plataforma IVVI de adquisición de datos independiente para la adquisición de datos sincronizada utilizando la arquitectura ROS. Es decir, ROS permite una colección de drivers y middleware que pretende simplificar la compleja tarea de la adquisición de datos y la sincronización del sensor. Además, los procesos de fusión sensor permiten para mejorar el rendimiento de cada aplicación en etapas de alto nivel.

Así, las aplicaciones presentadas en este trabajo se han creado en consecuencia para su futura aplicación en el sistema IVVI basado en ROS.

4.3.Software

A continuación, se exponen características y peculiaridades de los programas, librerías y paquetes de software utilizados en el proyecto.

4.3.1. QtCreator

QtCreator [22] ha sido el entorno de desarrollo o IDE utilizado en el proyecto. QtCreator es un entorno de desarrollo integrado multiplataforma para C++, JavaScript y QML. Usa el compilador de C++ de la colección de compiladores GNU en Linux. En Windows puede usar MinGW o MSVC con la instalación por defecto y también Microsoft Console Debugger cuando se compila desde el código fuente.

La elección de este entorno fue condicionada por el previo conocimiento del programa debido a trabajos anteriores. Además, su gran popularidad favorece la existencia de grandes cantidades de información sobre su funcionamiento junto con ROS y openCV.

4.3.2. OpenCV

OpenCV [23] es una biblioteca de funciones de programación multiplataforma dirigida principalmente a visión por computador en tiempo real. Originalmente fue desarrollada por el centro de investigación de Intel Nizhny Novgorod (Russia), y es mantenida en la actualidad por Itseez. La biblioteca de OpenCV esta publicada bajo licencia BSD y, por lo tanto, puede ser usada libremente en ámbitos tanto académicos como profesionales siempre y cuando se cumplan las restricciones que las licencias BSD requieren.

OpenCV soporta los lenguajes de programación C++, C, Python y Java, y soporta Windows, Linux, Mac OS, iOS y Android. Escrita en un C/C++ muy optimizado, puede aprovechar las ventajas del procesamiento en paralelo. Habilitado con OpenCL, puede aprovechar las ventajas de la aceleración de hardware de la plataforma de computación heterogénea subyacente. Ampliamente extendida por el mundo, tiene un numero de descargas estimado de 9 millones. Sus usos varían desde prospección de minas, mapas, robótica, medicina…

Dado que el objetivo de este proyecto es crear una aplicación que se basa en visión por computador, estas bibliotecas de código libre son ideales para su realización. Además, están perfectamente integradas con el marco de trabajo ROS.

4.3.3. PCL

Las Point Cloud Library (PCL) [24] son un conjunto de librerías de algoritmos de código abierto enfocadas a procesamiento de geometría 3D y tareas de procesamiento de nubes de puntos. Contiene algoritmos para estimación de características, reconstrucción de superficies, registros, y segmentación entre otros. Están escritas en C++ y publicadas bajo licencia BSD.

Estos algoritmos se usan, por ejemplo, para percepción en robótica, para filtrar *outliers* de datos con ruido, juntar nubes de puntos 3D, segmentar partes relevantes de una escena, extraer puntos característicos y calcular descriptores para reconocer objetos reales basados en su apariencia geométrica, o crear superficies a partir de nubes de puntos y visualizarlas, por mencionar algunos de ellos.

PCL es multiplataforma, y ha sido compilado exitosamente en Linux, MacOS, Windows, y Anroid/iOS. Para simplificar el desarrollo, PCL se divide en una serie de librerías de código menor, que pueden ser compiladas de forma independiente. Ésta modularidad es importante para distribuir las PCL en plataformas con restricciones computacionales o de memoria.

Una nube de puntos (*point cloud)* es una estructura de datos usada para representar una colección de puntos multidimensionales y es comúnmente utilizada para representar datos tridimensionales. En una nube de puntos 3D, los puntos usualmente representan las coordenadas geométricas X, Y, Z de la superficie representada. Cuando existe información de color, la nube de puntos se convierte en 4D.

Las nubes de puntos pueden ser obtenidas de sensores como cámaras estéreo, escáneres 3D, cámaras de tiempo de vuelo (time-of-flight cameras), o generadas con un programa de ordenador. PCL soporta de forma nativa interfaces OpenNI 3D, y por consiguiente puede adquirir y procesar datos de sensores como cámaras PrimeSensor 3D, the Microsoft Kinect, o Asus XTionPRO.

4.3.4. ROS

ROS [25] es un marco de trabajo para el desarrollo de software dirigido a robots. Consiste en una colección de herramientas, librerías, y convenciones que aspiran a simplificar la tarea de crear comportamientos robóticos complejos y robustos, sobre una amplia variedad de plataformas robóticas. ROS fue creado con el propósito de fomentar el desarrollo de software robótico colaborativo. Así expertos de todo el mundo pueden contribuir al desarrollo de este sistema, avanzando en sus respectivas áreas de conocimiento, y aprovechando el trabajo de otros investigadores.

ROS proporciona funciones y servicios similares a las de un sistema operativo en un clúster de ordenadores heterogéneo. Tiene abstracción del hardware, control de dispositivo a bajo nivel, implementación de funcionalidad de uso común, paso de mensajes entre procesos y mantenimiento de paquetes. Los procesos en ejecución basados en ROS son representados en una arquitectura de grafos, donde el procesamiento tiene lugar en nodos que pueden recibir, publicar y multiplexar mensajes de sensores, control, estado, planificaciones, actuadores y otros mensajes.

Las características de ROS permiten desarrollar el proyecto facilitando una gran cantidad de tareas. Al abstraerse del hardware cabe la posibilidad de actualizar el equipo de cámaras del coche, y aun así mantener el mismo código sin necesidad de modificarlo. Además, tal y como ha sido comentado anteriormente, este proyecto podrá ser guardado y compartido en un paquete, que podrá ser usado por el resto del departamento, o incluso de la comunidad investigadora.

Como consecuencia de las características de ROS, el programa constará de una serie de nodos, que se comunicarán entre sí a través de topics. Cada nodo podrá suscribirse (obtener información) o publicar (mandar información) en dichos topics, diferentes mensajes estandarizados.

5. Estructura del programa

En este capítulo se resume la estructura interna de la aplicación final basada en ROS. En este apartado también se empieza a introducir el trabajo realizado por el alumno, tanto el de desarrollo del código de los paquetes nuevos, como el de integración de los distintos paquetes de existentes. Tal y como se vio en la sección 4.3.4, el procesamiento de la aplicación final tiene lugar en nodos. Estos nodos junto con la comunicación que se produce entre ellos, se explican brevemente para proporcionar al lector una idea del funcionamiento del programa.

El programa se divide en 6 partes o nodos como consecuencia del modo de funcionamiento de ROS. En un futuro, esto permitirá actualizar o mejorar ciertas partes del código de alguno de estos nodos, y mantener la estructura del programa en perfecto funcionamiento.

En primer lugar, un nodo capta las imágenes de la cámara y algunas características del hardware que serán necesarias posteriormente. Inmediatamente después, estas imágenes se publican en un topic. El siguiente nodo, el cual fue desarrollado por el LSI-Laboratorio de Sistemas Inteligentes, leerá estas imágenes del topic al que está suscrito, y las rectificará, publicándolas posteriormente bajo un topic con distinto nombre.

A partir de este momento, se empieza a ejecutar la parte de código desarrollada en este proyecto.

El siguiente nodo calcularía el mapa de disparidad, con alguno de los algoritmos ejecutados en GPU estudiados anteriormente. A continuación, otro nodo subscrito al topic donde se publican las disparidades, creará una nube de puntos a partir de este mapa de disparidades. Finalmente, otro nodo calculará la odometría necesaria para una correcta visualización y la creación de un modelo 3D preciso. El nodo link1_broadcaster simplemente se encarga de publicar una transformación geométrica o *tf* entre el robot y la cámara. Esta relación se explicará con más detalle en el apartado 8.1 Odometría visual.

En la Figura 27 se muestra la relación entre los nodos. Las elipses representan los nodos del programa y las flechas los topics sobre los cuales se comunican estos nodos. En la Figura 28 se muestra la relación entre nodos del mismo programa, pero con el nodo extra voxel_grid [26] que reduce la cantidad de puntos mejorando el uso de memoria en la visualización.

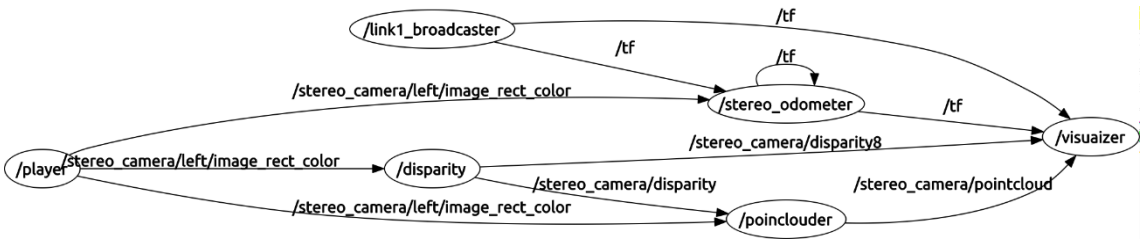


Figura 27: Relación entre nodos y correspondientes topics del programa implementado en ROS

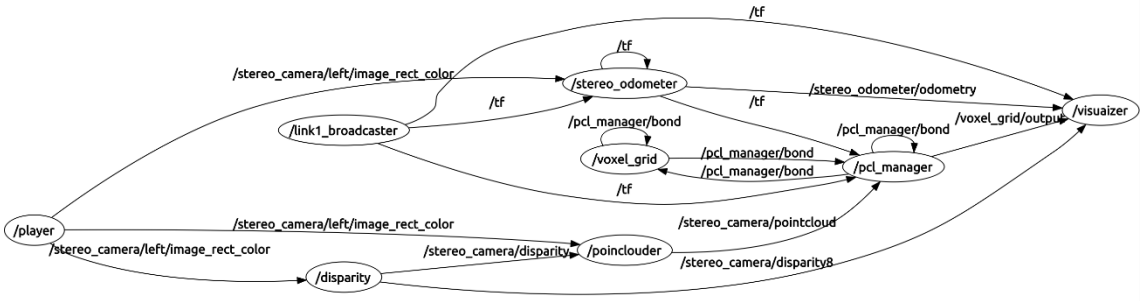


Figura 28: Relacion entre nodos con voxel_grid incluido

La Figura 29 representa un diagrama de flujo del programa.

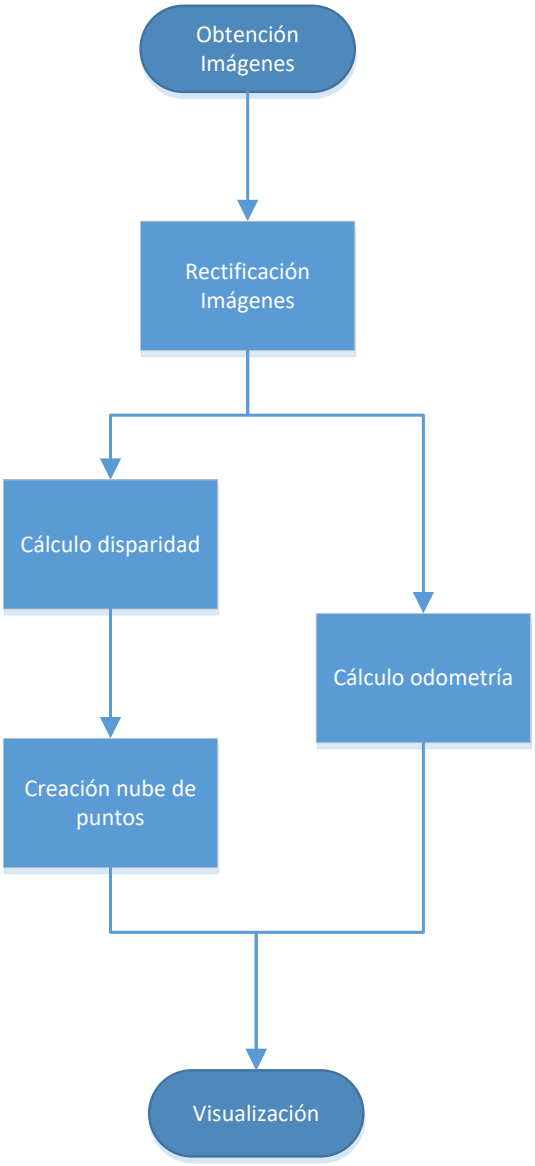


Figura 29: Diagrama de flujo del programa

6. Construcción del mapa de disparidad

En primer lugar, antes de considerar la aplicación en nuestro proyecto de alguno de los algoritmos de cálculo de disparidad ejecutados en GPU que ofrecen las OpenCV como *stereoBM* [27] *stereoBeliefPropagation* [28] o *StereoConstantSpaceBP* [29], había que comprobar si verdaderamente eran más eficientes que los algoritmos ejecutados en CPU stereoBM [27] y stereoSGBM [30], también de la biblioteca de las OpenCV. Sin comprobar que efectivamente existía una mejora en el rendimiento, no tenía sentido la continuación del proyecto con el uso de estos nuevos algoritmos.

Con este propósito se propuso crear un algoritmo que obtuviese mapas de disparidad del mismo set de imágenes utilizando los distintos métodos. De esta manera, si más tarde se cuenta con unas medidas "exactas" de la escena en el mundo real, se podría comparar la precisión de estos métodos. También, sería posible obtener más medidas objetivas de eficiencia como tiempo de ejecución para cada método, densidad de pixeles, media del error absoluto etc.

Gracias a los centros de *Karlsruhe Institute of Technology* y Toyota *Technological Institute at Chicago* y a su base de datos "*The KITTI Vision Benchmark Suite*" [31] se pudo realizar esta comparación.

Esta base de datos consta de cientos de imágenes tomadas con cámaras estéreo en diversos tipos de carreteras y situaciones. No obstante, el motivo principal para la elección de esta base de datos fue debido a la existencia de mediciones precisas para cada imagen tomadas con un escáner láser Velodyne LiDAR. Estas mediciones tomadas con el escáner laser ofrecen una medición precisa en mundo real (*ground truth)* de las distancias que se observan en la imagen. Con este *ground truth* se puede cuantificar el error que se obtiene de cada método, y compararlos fácilmente.

Era crucial realizar este estudio antes de continuar con la reconstrucción en 3D por dos motivos principalmente.

La calidad de la futura reconstrucción tridimensional depende de la precisión de este mapa de disparidades. Por ello se realiza una comparación de estos algoritmos tomando como referencia el ground truth. No obstante, como este sistema está diseñado para ejecutarse en una plataforma embebida(las cuales tienen con una velocidad de procesamiento reducida) y aspira a funcionar en un sistema a tiempo real, la velocidad de procesamiento de estos algoritmos es, como mínimo, igual de importante que la precisión.

Estos dos factores, velocidad y precisión determinaran la elección final del algoritmo a realizar.

A continuación, se muestra un ejemplo con una imagen aleatoria para clarificar los distintos tipos de imágenes que intervienen en el proceso.

En la Figura 30 se observa una furgoneta y un coche circulando por una calle bidireccional y con gran cantidad de coches aparcados a ambos lados. Complementando esta

imagen en concreto, obtenida de la base de datos KITTI, se puede observar su respectivo ground truth en la Figura 31. Este ground truth está almacenado en un formato de tipo imagen, en donde cada punto que se representa en la imagen, corresponde con una medición del mundo real realizada con el escáner laser Velodyne. Los pixeles nulos o en negro, significan que no hay información del escáner Velodyne para ese punto. Los pixeles que aparecen en un tono de gris, son puntos en los que el escáner laser hizo una medición, y su intensidad va en función de la distancia. Cuanto más lejos está un objeto, pongamos de ejemplo la furgoneta que va circulando, menor intensidad tendrá el tono de gris, o más cerca del negro estará ese pixel. Cuanto más próximo este un objeto, pongamos de ejemplo el coche que se acerca hacia el observador, mayor intensidad tendrá el tono de gris, o más cerca del blanco estará ese pixel.

Figura 30: Imagen izquierda del KITTI data set [31]

Figura 31: Ground truth de la imagen superior [31]

Como se ha mencionado con anterioridad, este proyecto utiliza una cámara estéreo, por lo que de cada escenario existe una imagen izquierda y otra derecha.

Figura 32: Imagen derecha del KITTI data set [31]

Gracias a la diferencia entre ambas imágenes, es posible crear un mapa de disparidad en el que quede representado la profundidad de este escenario. Un ejemplo sería la Figura 33, donde los pixeles más intensos, o más cerca del blanco, representan una mayor proximidad al observador.

Figura 33: Mapa de disparidad de la imagen superior con el método ConstantSpaceBP

La Figura 34 muestra el mismo mapa de disparidad usando gradiente de color (colores cálidos indican mayor proximidad, y fríos menor proximidad al observador), superpuesto a la imagen izquierda original.

Figura 34: Imagen original superpuesta al mapa de disparidad representado en gradiente de colores (colores cálidos más próximos y colores fríos más lejanos)

Las distintas funciones para calcular el mapa de disparidad tienen diversos parámetros que modifican el resultado final. Pero hay dos atributos principales que se deben tener en cuenta, el *SADWindowSize* o tamaño de bloque y el *ndisparities* o número de disparidades.

El número de disparidades indica el rango de búsqueda de disparidad. Para cada pixel, el algoritmo buscara la mejor disparidad desde 0 hasta la disparidad máxima indicada.

El tamaño de bloque o *SADWindowSize* indica el tamaño de los bloques comparados por el algoritmo (por ejemplo, 3x3 centrado en el pixel a examinar). Un mayor tamaño de bloque genera un mapa de disparidades con menos ruido, aunque menos preciso. Un menor tamaño de bloque genera un mapa de disparidades más detallado, pero la probabilidad de que el algoritmo encuentre una disparidad errónea es mayor.

6.1. Método de medición del error

Para poder medir con precisión la calidad de estos mapas de disparidad, se creó una función llamada `calculate_disp_err` la cual cuantificaba la calidad de los mapas de disparidad obtenidos. Esta función se basa en una función ya existente creada para Matlab, también ofrecida por KITTI en un *development kit*.

El proceso para el cálculo del error en el mapa de disparidad sigue la siguiente fórmula:

$$E(i,j) = \left|D_{gt}(i,j) - D_{est}(i,j)\right|$$

$$err(i,j) = D_{gt} > 0 \ \wedge \ E(i,j) > \tau_1 \ \wedge \ \frac{\mathrm{E}(i,j)}{\left|\mathrm{D_{gt}}(i,j)\right|} > \tau_2$$

$$n_{err(i,j)} = \sum [err(i,j)]$$

$$d_{err} = \frac{n_{err}}{n_{total}}$$

En donde:

- $\boldsymbol{D_{gt}}$ es la imagen del ground truth.
- $\boldsymbol{D_{est}}$ es la imagen del mapa de disparidad obtenido.
- $(\boldsymbol{i},\boldsymbol{j})$ indica el indice en la matriz de la imagen (señala a un pixel en concreto).
- $\boldsymbol{E}$ es el error entre el valor de la disparidad en un pixel dado y su correspondiente ground truth.
- $\boldsymbol{err}$ es un valor booleano que indica si para el pixel considerado, se cumplen todas las condiciones expuestas.
- $\boldsymbol{\tau_1}\ y\ \boldsymbol{\tau_2}$ son dos parámetros para evaluar la existencia de error (uno absoluto y otro relativo). Para este proyecto, se utilizaron los valores recomendados por KITTI $\boldsymbol{\tau_1} = 3\ y\ \boldsymbol{\tau_2} = 0.05$.
- $\boldsymbol{n_{err}}$ es el número de errores existentes el mapa de disparidad calculado.
- $\boldsymbol{n_{total}}$ es el número total de valores definidos en el ground truth.
- $\boldsymbol{d_{err}}$ es el error de disparidad en tanto por uno.
- El operador **∧** representa la operación booleana AND, y **[·]** es la función indicadora del corchete Iverson que genera 1 cuando la declaración en cuestión es verdadera, y 0 en el caso contrario.

Básicamente, se calcula la diferencia absoluta entre los valores de intensidad de pixel de las imágenes de ground truth y mapa de disparidad. En los puntos donde el ground truth no está definido, no se realiza esta operación.

Dependiendo de si esta diferencia es mayor que un cierto valor τ_1, y a su vez el error relativo con respecto al valor del ground truth es superior a τ_2, se considera la existencia de un error en el cálculo de la disparidad y se contabiliza como tal.

El parámetro τ_2 se aplica para que no se contabilicen como un error aquellos casos en los que la diferencia absoluta entre la imagen y el ground truth es mayor que τ_1, pero el error relativo con respecto al valor original de ground truth es pequeño. Por ejemplo, el error calculado puede tener un valor de 10, pero si el valor del ground truth es mayor de 200, el error porcentual es inferior al 5%.

En los puntos en donde el valor del ground truth está definido, pero el mapa de disparidad no lo está, también se considera que existe error en el cálculo de la disparidad.

Finalmente se suman todos los errores contabilizados y se dividen por el número total de pixeles definidos en el ground truth. Esto genera un error porcentual del mapa de disparidades.

A continuación, se ofrece un análisis cualitativo de los diversos métodos disponibles en las OpenCV. En la sección 9.1 se reflejará el correspondiente análisis cuantitativo.

6.2. Método CPU stereoBM

En este apartado y en los consecutivos, se razona la selección que se hizo del algoritmo final y se muestran los distintos tipos de mapas de disparidades. En un intento de hacer una comparación coherente, se determinó el mismo número de disparidades (96, salvo excepciones claramente explicadas) para todos los algoritmos. Concretamente los parámetros de la cámara con la que se realizaron las fotos del banco de datos KITTI son: distancia focal $f = 707.0912\, pixeles$ y baseline $B = 0.537904\, m$. Con estos valores, la profundidad Z que corresponde al número de disparidades elegido, que en este caso han sido 96, sería $Z = \frac{f*B}{d} = \frac{707*0.54}{96} = 3.98\, m$. Este valor sería la Z mínima que aparecería en los mapas de profundidades, lo cual no supone ningún problema ya que los objetos de la escena están por lo general a una distancia mayor.

Para una mayor visualización, se ha elegido mostrar solo una fotografía elegida al azar, que será recurrente en todos los algoritmos (Figura 35).

Figura 35: Imagen de la base de datos KITTI (nº 000126) [31]

El método BM es el más básico para calcular la disparidad de una imagen. Es sin duda el método más rápido, pero también el que genera los mapas de disparidad con peor calidad.

Figura 36: Mapa de disparidad con el método BM (ndisparities=96)

El mayor problema de este algoritmo es que tiene demasiadas zonas de las que no consigue sacar la disparidad, y quedan en negro. Se muestra el resultado en la Figura 36.

En general los algoritmos de visión estéreo sufren de varios problemas que pueden resultar en mapas de disparidad de baja calidad. Los problemas que más afectan a esta estimación de la disparidad son:

- Variación fotométrica
  - Para el mismo punto del mundo, su proyección en las cámaras tendrá valores distintos
- Oclusiones
  - Existen discontinuidades en la distancia
    - Una parte del mundo solo se ve en una cámara
    - Si se analiza una ventana se tiene objetos a distintas distancias
- Textura repetitiva
  - Hay varias soluciones buenas
- Falta de textura
  - Hay varias soluciones malas

Es debido a estos problemas por lo que el mapa de disparidades estimado tiene zonas en negro en donde no se ha conseguido obtener una estimación válida.

A la hora de introducir este mapa de disparidades en el algoritmo de medición del error, se generan gran cantidad de errores, debido a que el algoritmo detecta como fallo los pixeles en donde no existe información de disparidad (y existe información del escáner en el ground truth). Tendríamos así, fallos debido a un cálculo erróneo en la disparidad (difiere demasiado con el ground truth) y puntos no calculados, consiguiendo que este método obtenga la peor posición en los resultados de la comparativa.

6.3. Método CPU stereoSGBM

Este método consigue generar un mapa de disparidad de mucha mayor calidad que el método anterior, observable en la Figura 37, pero su tiempo de ejecución también aumenta en gran medida.

Figura 37: Mapa de disparidades método SGBM (ndisparities=96)

Se obtiene una densidad de pixeles mucho más elevada, y como consecuencia se generan menos errores por falta de información.

6.4. Método GPU stereoBM

Este método es la adaptación a CUDA del método stereoBM visto anteriormente. No obstante, dentro del mismo método se realiza un post-procesamiento de la imagen, que consigue un mapa de disparidades con una densidad de pixeles mucho más elevada. Se puede apreciar la diferencia con el método BM de CPU en la Figura 38.

El tiempo de ejecución aumenta, pero teniendo en cuenta que se consigue un mapa de disparidades de mayor calidad, y que el aumento del tiempo de ejecución no excede los límites de nuestra plataforma embebida, se considera que este método cumple mejor con los requisitos que el implementado en CPU.

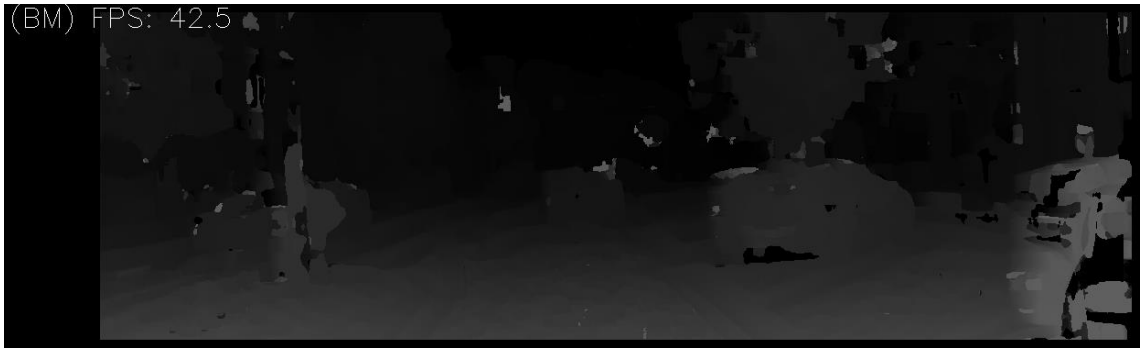


Figura 38: Mapa de disparidades método BM en CUDA (ndisparities=96)

Para una visualización más cómoda, se le ha dado color a la imagen, siendo los colores cálidos objetos más cercanos, y los colores fríos más lejanos (Figura 39):

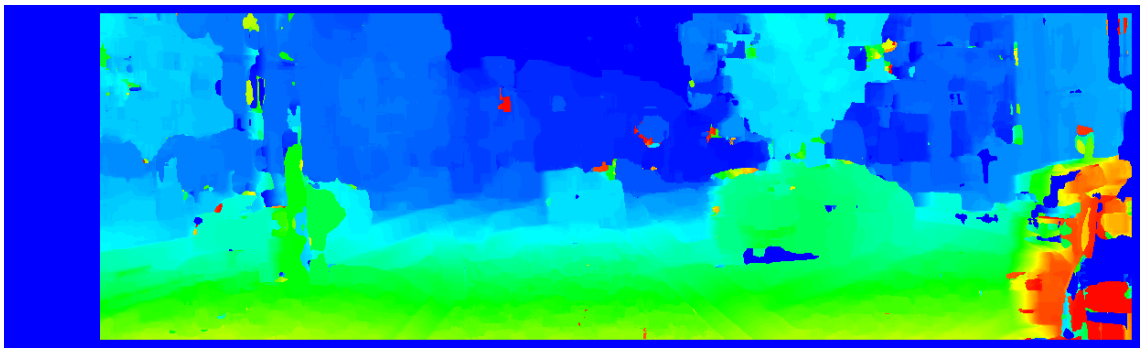


Figura 39: Mapa de disparidades color método BM en CUDA (ndisparities=96)

6.5. Método GPU stereoBeliefPropagation

El método de *Belief Propagation* no es adecuado para una plataforma embebida ya que tiene una velocidad de procesamiento pésima con respecto a los otros algoritmos. Incluso el mapa de disparidad obtenido no presenta la calidad que se podría esperar viendo el tiempo necesario para procesarlo (Figura 40).

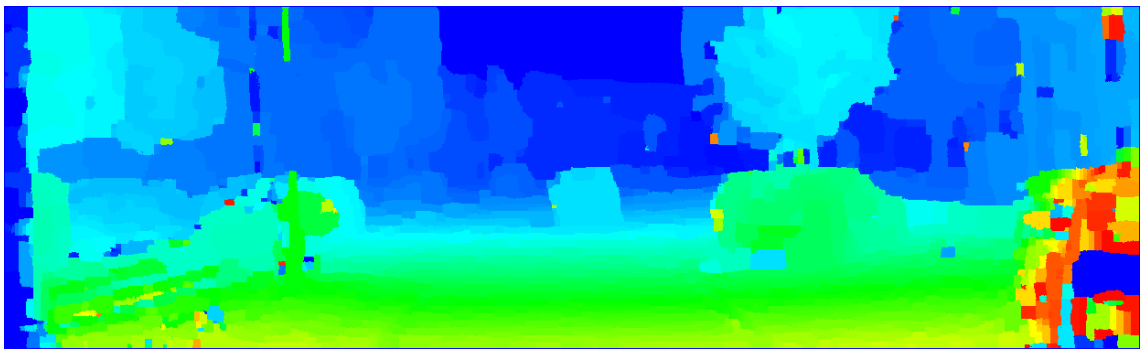


Figura 40: Mapa de disparidades método BP (ndisparities=96)

Para este método, existe una función en las bibliotecas OpenCV llamada `estimateRecommendedParams`. Esta función utiliza un modelo heurístico que estima los parámetros necesarios para el algoritmo (como puede ser el número de disparidades, iteraciones, o de niveles) en función del tamaño de la imagen original. Se utilizó esta función para ver los resultados, pero inexplicablemente con la aplicación de los nuevos parámetros estimados, el tiempo de procesamiento aumento en más de dos veces y media con respecto al cálculo anterior, no haciéndolo en la misma medida la precisión del nuevo mapa de disparidades estimado.

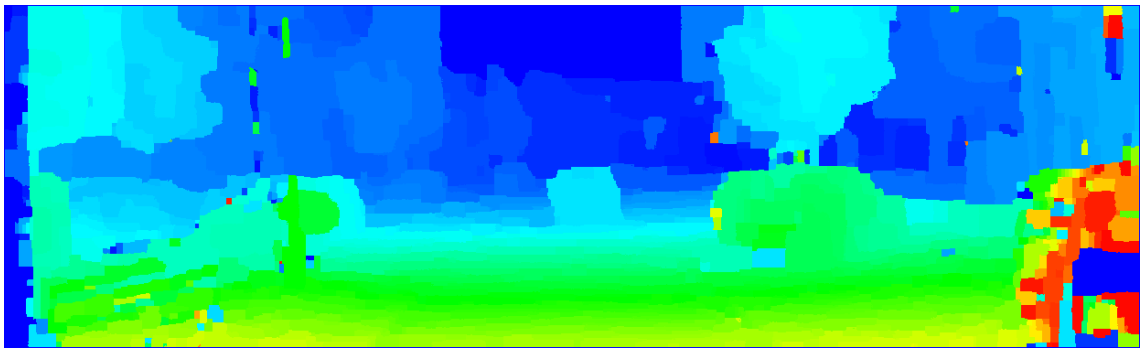


Figura 41: Mapa de disparidades método BP parámetros estimados

En la sección 9.1 Comparativa entre CPU/GPU (Tabla 1 y Tabla 2), se observa que el tiempo de procesamiento es mucho mayor que los otros algoritmos, mientras que el mapa de disparidad no mejora en la misma proporción.

6.6. Método GPU stereoConstantSpaceBP

El método stereoConstantSpaceBP o CSBP, es sin duda el que mejor rendimiento ofrece.

Con una velocidad de procesamiento de aproximadamente la tercera parte que el SGBM, consigue obtener mapas de disparidad de prácticamente igual calidad. Además, cuenta con una densidad de pixeles prácticamente del 100%, lo cual es una gran ventaja a la hora de generar nubes de puntos en la siguiente fase. El resultado se muestra en la Figura 42.

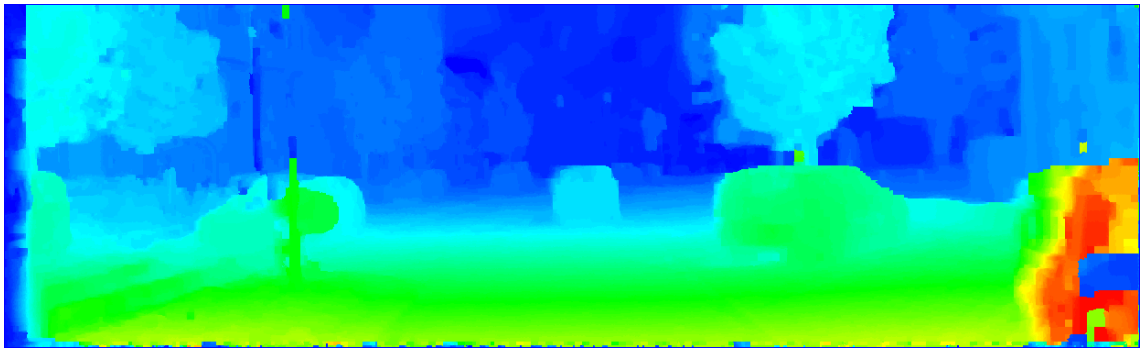


Figura 42: Mapa disparidad método CSBP ndisparities=96

Al igual que en el caso anterior (BeliefPropagation), se utilizó la función `estimateRecommendedParams`. Aunque en este caso sí se obtiene un mapa de disparidades mucho más rápidamente, reduciéndose el tiempo de ejecución a casi la mitad. Los resultados tras la aplicación de esta función se muestran en la Figura 43 y Figura 44:

Figura 43: Mapa disparidad método CSBP parámetros estimados

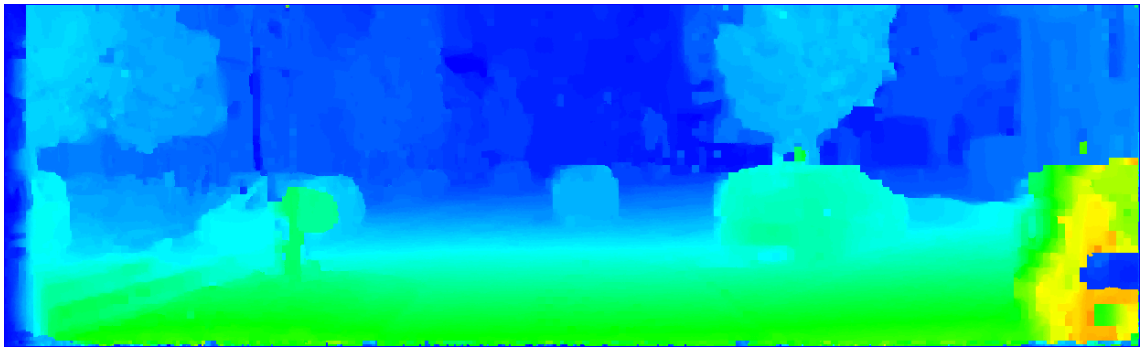


Figura 44: Mapa disparidad método CSBP parámetros estimados

En esta imagen, calculada con los parámetros estimados, parece que la gama de colores es más fría o que estos los objetos están más lejanos en comparación a la imagen anterior. Esto es debido a que la representación de la profundidad en función de los colores, en la función usada, va en función de la variable número de disparidades. En los mapas de disparidad calculados con los parámetros estimados, el número de disparidades máximas son de 120 para esta imagen (ejecutada en el portátil ASUS), produciendo que los valores de intensidad de cada pixel en el mapa de disparidad parezcan menores en su conjunto.

Figura 45: Imagen original fundida con el mapa de disparidad del método CSBP parámetros estimados

Después de estudiar los resultados obtenidos en la comparativa de los distintos métodos, se llegó a la conclusión de que el más adecuado a las características de una plataforma embebida como la de este proyecto era el CSBP.

Finalmente se elige este método de cálculo de mapa de disparidades para continuar con la siguiente fase del proyecto, la generación de la nube de puntos. En la Figura 45 se muestra una superposición entre la imagen original y el método CSBP elegido finalmente.

7. Creación de nube de puntos en 3D

Una vez obtenido el mapa de disparidad y siguiendo las ecuaciones vistas en la sección 3.2.2, se puede calcular el mapa de profundidades o *depth map*. Se le asigna a cada pixel del mapa de disparidades un nuevo valor que representa en una unidad de medida de longitud la distancia del objeto al observador.

Contando con estas profundidades o *depth map*, lo único que hay que hacer es calcular la localización de cada objeto en el mundo real. Se construye una nube de puntos utilizando las librerías de código abierto PCL y sirviéndose de este *depth map* para la coordenada Z (profundidad), y con las ecuaciones vistas en la sección 3.1 para calcular la X y la Y (plano de la cámara). La nube de puntos creada es una estructura que representa las coordenadas Euclídeas XYZ, así como información de color RGB de cada punto. Como consecuencia, se obtiene para cada pixel un punto con coordenadas XYZ y color RGB que se almacena en esta estructura de nube de puntos.

Este proceso de obtención de la nube de puntos correspondiente a un mapa de disparidad se realizó con la función de las PCL 1.8 `pcl::cuda::DisparityToCloud::compute`. Esta función aprovecha el poder de la GPU para calcular el *depth map* (Z), las coordenadas XY, y almacenar toda la información (XYZ-RGB) en una estructura de nube de puntos, como la que se puede observar en la Figura 46 y Figura 47.

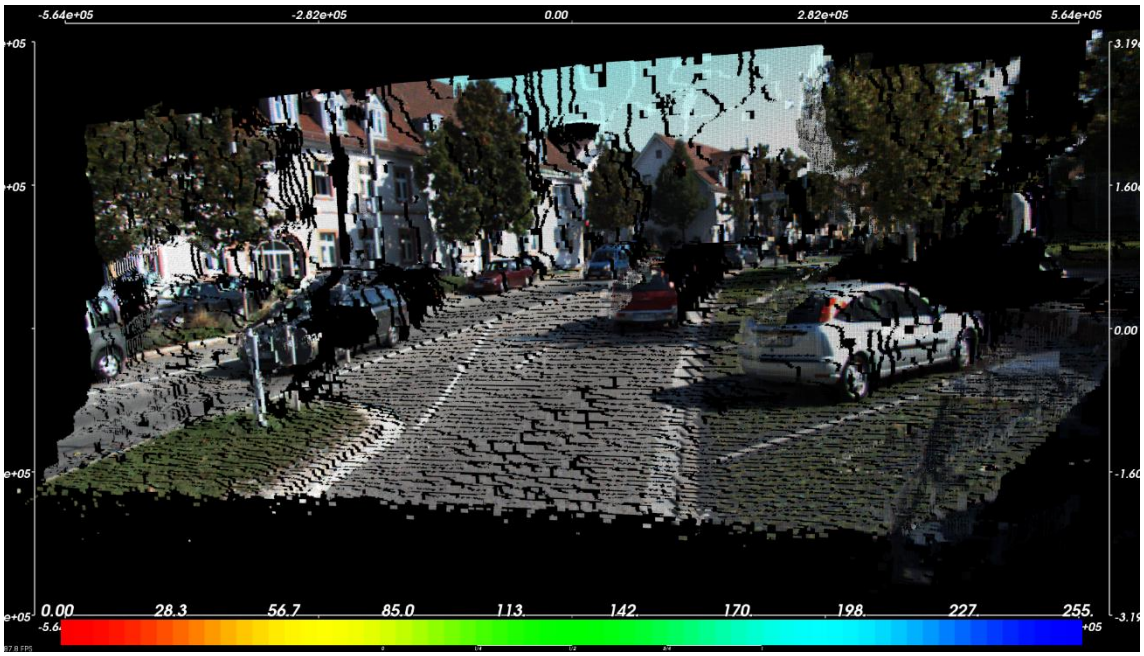


Figura 46: Nube de puntos obtenida de la reconstrucción del mapa de disparidad anterior.

Actualmente en las PCL, la forma de estructurar las nubes de puntos difiere entre las implementadas en CPU y GPU, existiendo dos clases distintas. Por ello, al haber realizado el cálculo en la GPU, es necesario hacer una copia de memoria y transformar la estructura a una

nube de puntos en la CPU. Esto es necesario ya que las nubes generadas van a ser leídas por un paquete de ROS, y éste espera recibir la nube de puntos en la estructura clásica de CPU.

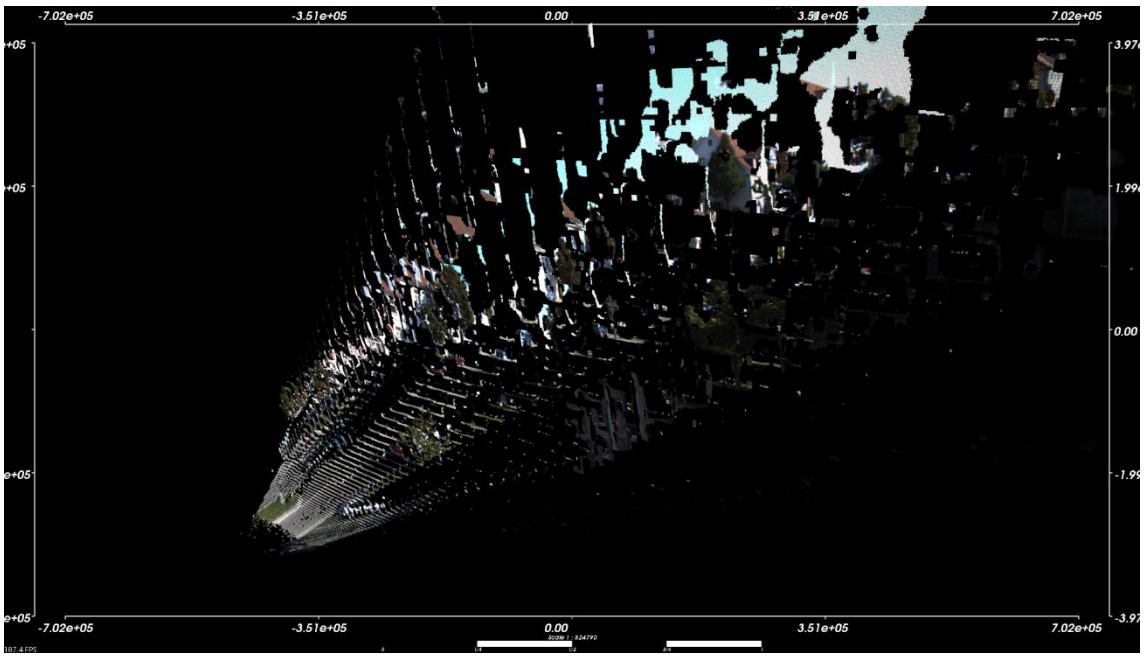


Figura 47: Misma nube de puntos que la Figura 46 desde otra perspectiva. Las múltiples "líneas" vacías de información son debidas a la función utilizada. Las variables aceptadas son de tipo integer, o numero entero. La falta de continuidad entre capas o planos se debe al salto que existe entre dos números enteros.

Como ya se ha comentado anteriormente, el objetivo de este proyecto era el de sacar el máximo partido a la GPU. Por ello, tanto el cálculo del mapa de disparidad como el cálculo de la nube de puntos se realizó en CUDA sobre la tarjeta gráfica. En la Figura 49 se muestra la parte de código que se encarga de la construcción de la nube de puntos. Las coordenadas (X,Y,Z) son obtenidas con la geometría vista en la sección 3.1 (coordenadas X,Y), y con la ecuación vista en la sección 3.2.2 $Z = f\frac{B}{d}$ (para la coordenada Z). Cada una de las coordenadas propias de un punto se calcula en un hilo o *thread*. Lanzando un hilo para cada punto se consigue aumentar la velocidad en gran medida.

```
template <typename Tuple> PointXYZRGB
ComputeXYZRGB::operator () (const Tuple &t)
{
  PointXYZRGB pt;

  if (thrust::get<0>(t) == 0) // assume no_sample_value and shadow_value are also 0
    pt.x = pt.y = pt.z = bad_point;
  else
  {
    // Fill in XYZ (and copy NaNs with it)
    pt.x   = ((thrust::get<2>(t) % width) - center_x) * thrust::get<0>(t) * constant * 0.001f;
    pt.y   = ((thrust::get<2>(t) / width) - center_y) * thrust::get<0>(t) * constant * 0.001f;
    pt.z   = thrust::get<0>(t) * 0.001f;
  }
  pt.rgb = thrust::get<1>(t).r << 16 | thrust::get<1>(t).g << 8 | thrust::get<1>(t).b;
  return (pt);
}
```

Figura 48: Parte de la función `disparity_to_cloud.cu` en donde se crea la nube de puntos

La clase de las librerías PCL 1.8 DisparityToCloud se tuvo que modificar ya que, en su implementación a la hora de crear la nube de puntos (Figura 48), la función espera directamente un mapa de profundidades. Se decidió que era más eficiente asignarle el mapa de disparidades y que se realizase el cálculo del mapa de profundidades ya en la GPU, utilizando un hilo por punto.

En la Figura 49 se muestra la parte del código modificado en la función del cálculo de la nube de puntos. La variable `constant` no es más que la inversa de la distancia focal f. Finalmente, se quita el factor de escala `0.001`, cuyo objetivo era el de la conversión del `baseline` a metros. Los nodos de ROS que ofrecen la información de la cámara ya dan estas medidas en metros y no es necesario realizar esta conversión.

```
template <typename Tuple> PointXYZRGB
ComputeXYZRGB::operator () (const Tuple &t)
{
  PointXYZRGB pt;

  if (thrust::get<0>(t) == 0) // assume no_sample_value and shadow_value are also 0
    pt.x = pt.y = pt.z = bad_point;
  else
  {
    // Fill in XYZ (and copy NaNs with it)
    pt.z   = baseline / (constant * thrust::get<0>(t));
    pt.x   = ((thrust::get<2>(t) % width) - center_x) * pt.z * constant;
    pt.y   = ((thrust::get<2>(t) / width) - center_y) * pt.z * constant;

  }
  pt.rgb = thrust::get<1>(t).r << 16 | thrust::get<1>(t).g << 8 | thrust::get<1>(t).b;
  return (pt);
}
```

Figura 49: Modificación del código en Figura 48

8. Odometría visual, acumulación de nubes de puntos y visualización en ROS

Finalmente se aborda el problema de la implementación de este programa en ROS. Se crean nodos para el cálculo de disparidad utilizando el método CSBP en GPU, para la creación de nube de puntos en GPU con las librerías modificadas PCL 1.8, y se usan los nodos ya existentes de rectificación de la imagen desarrollado en el departamento, el cálculo de odometría incluido en el paquete *viso2_ros* y el visualizador *rviz*.

8.1. Odometría visual

El paquete de ROS *viso2_ros, implementado por S. Wirth en la UIB, basándose en el algoritmo StereoScan de Geiger, Ziegler y Stiller* [31] contiene dos nodos, `mono_odometer` y `stereo_odometer`. Ambos estiman el movimiento de la cámara basándose en las imágenes rectificadas provenientes de una cámara calibrada. Para estimar la escala del movimiento, la odometría 'mono' usa el plano del suelo y por consiguiente necesita información acerca de la coordenada Z de la cámara y de su ángulo de elevación o pitch. La odometría estéreo, la cual es usada para este proyecto, no necesita de ningún parámetro adicional.

En ROS existe un paquete muy común especializado en transformaciones geométricas entre sistemas de coordenadas llamado *tf*. Haciendo uso de transformadas *tf*, el paquete viso2_ros es capaz de generar transformadas entre el sistema de referencia mundo, el cual correspondería con el inicio de la secuencia, y el sistema de coordenadas situado en la cámara, el cual correspondería con la posición actual del vehículo.

La cadena de transformaciones geométricas relevantes para la odometría visual es la siguiente:

$$mundo \rightarrow odom \rightarrow base\_link \rightarrow cámara$$

Generalmente los algoritmos de odometría visual calculan el movimiento de cámara. Para poder realizar el cálculo de movimiento del robot (vehículo en el caso del proyecto) basándose en el movimiento de la cámara, ha de conocerse la transformación geométrica desde la cámara al robot. Por ello, esta implementación necesita conocer la transformada geométrica o *tf* entre *base_link* (base del robot) y la cámara. En este caso, se ha decidido omitir esta transformación y asumirla como 0 (base_link y cámara son coincidentes), ya que es irrelevante la posición relativa de la cámara con respecto al sistema de referencia del vehículo para el objetivo de este proyecto.

En la Figura 50 se muestra un ejemplo de un posible escenario hipotético, en el que un vehículo con un escáner laser se va desplazando por el espacio, y su relación al sistema de referencia *map*. Para este caso, el paquete viso2_ros generaría la transformación geométrica entre los sistemas de coordenadas odom y laser_link.

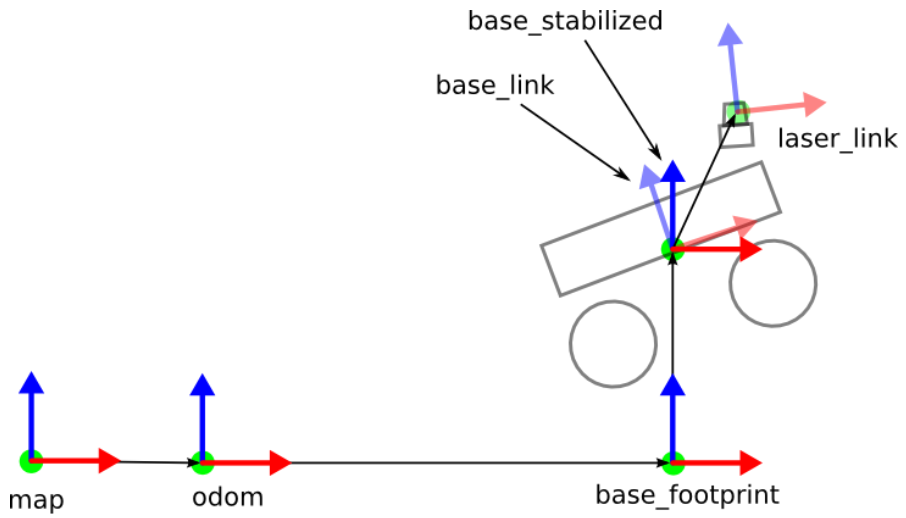


Figura 50: Sistemas de coordenadas en un escenario, y su relación entre ellos. [32]

8.2. Acumulación de nubes de puntos y visualización

Para la visualización final de las nubes de puntos se utilizó el paquete de ROS *rviz*. Rviz es un entorno de visualización 3D pensado en la depuración de aplicaciones robóticas. En este paquete se realizan una serie de configuraciones en donde se asigna al nodo una serie de topics a los que subscribirse, con el fin de obtener una visualización correcta del entorno.

Estos topics pueden ser la nube de puntos obtenida para un fotograma, la odometría calculada, las imágenes originales rectificadas o el mapa de disparidad en formato de 8bit.

Se puede observar en la sección 9.2 la escena reconstruida en 3D tras la correcta configuración de rviz. Por falta de textura en el cielo, el algoritmo CSBP comete un error de estimación en el mapa de disparidad, en la esquina superior derecha de la imagen. Esto genera una gran cantidad de puntos que están situados en una zona del espacio equivocada. Este problema se logra enmascarar aplicando un filtro en el eje 'Y' y eliminando esos puntos en la visualización, pero es un problema que habría que considerar en un futuro.

En la visualización de rviz, se aprecia el avance del vehículo por el espacio, calculado por el nodo de la odometría (flechas rojas), y se realiza una acumulación de puntos por un periodo de tiempo, antes de que estos desaparezcan para liberar memoria

Se ejecuta finalmente un nodo llamado voxel_grid que se encarga de reducir el tamaño de cada nube de puntos generada de cada mapa de disparidad, con el fin de mejorar el uso de memoria y que el programa no se sature con la cantidad de puntos acumulados. Este filtro reduce la densidad de la nube de puntos, además de acotar la distancia valida de representación (por ejemplo, solo se representan puntos que estén en el rango de 2 a 30 metros del vehículo). Un ejemplo puede verse en la sección 9.2, en donde se aprecia claramente la reducción de la densidad de la reconstrucción realizada.

9. Resultados

Los objetivos de este proyecto se resumen en dos. Primero, el estudio de los métodos de cálculo de disparidad disponibles y su comparación en cuanto a rendimiento, y, en segundo lugar, el uso de uno de estos algoritmos para realizar una reconstrucción de la escena en 3D. Es por este motivo que este apartado se divide en dos. Uno mostrará la comparativa realizada y el otro mostrará el resultado final de la reconstrucción.

9.1. Comparativa entre CPU/GPU

La calidad del mapa de disparidades es vital para una precisa reconstrucción del entorno. Por ello, se dedicó mucho tiempo y esfuerzo a analizar los métodos disponibles en las librerías OpenCV.

En este proyecto se busca conseguir un sistema integrado o "autocontenido", es decir, reducir el espacio al máximo y conseguir un sistema funcional que realice su función. Por ello además de una calidad del mapa de disparidades alta, es imprescindible que la velocidad de procesamiento sea lo suficientemente alta como para poder tener un sistema que llegue a funcionar casi a tiempo real.

Para el estudio se utilizaron las 200 imágenes que se consiguieron del banco de datos KITTI, con un tamaño de 1242x375. Tras el procesamiento de cada una de estas imágenes, los parámetros como el tiempo transcurrido, el porcentaje de error en la estimación del mapa de disparidades, o la densidad de pixeles son guardados en un fichero de texto.

Este fichero se importa a una tabla de Microsoft Excel en la cual se organiza y se crean las tablas que se ven en la Tabla 1 y en la Tabla 2. Aquí se calcula el valor medio y la mediana de todos los parámetros calculados en la métrica del error que se introdujo en la sección 6.1, para cada una de las imágenes analizadas. La mediana se calcula para observar la desviación que tienen unos valores de otros. Dado que para cada pixel de la imagen que tiene un ground truth correspondiente existe un valor para el parámetro E, los parámetros visibles en la tabla de Excel *E_media* y *E_mediana* son los resultantes de aplicar la media y la mediana respectivamente a todos los valores E de cada imagen. Por tanto, para cada imagen se tiene un parámetro E_media y un parámetro E_mediana.

Finalmente, para obtener una mejor visualización de los resultados, se generan los gráficos observables en Figura 52 y Figura 53.

Después de analizar estos datos se llegan a varias conclusiones:

I. El algoritmo BeliefPropagation es demasiado costoso computacionalmente como para poder plantearse usarlo en tiempo real. Además, la precisión del mapa de disparidad estimado para este tipo de secuencias no corresponde a la calidad esperada tras tiempos tan prolongados de computo.

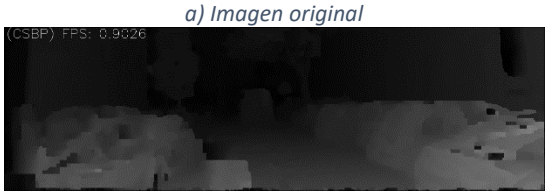
II. El algoritmo BlockMatching o BM implementado en CUDA es un gran candidato para poder ejecutarse en tiempo real. Con una tasa de error aceptable, es el algoritmo más rápido en computar un mapa de disparidades.

III. El algoritmo CSBP implementado en CUDA es el otro gran candidato para poder utilizarse en tiempo real. En situaciones donde sea necesario una mayor precisión, se observa que este método tiene una ventaja con respecto al método BM. Además, se ha comprobado que ajustando los parámetros de éste algoritmo (como numero de disparidades, iteraciones niveles…etc.), se consigue una velocidad de procesamiento menor, muy cercana a la conseguida con el método BM implementado en CUDA.

En la Figura 51 se muestran las distintas imágenes relevantes en esta comparativa. Se muestra la imagen original, su ground truth correspondiente y el mapa de disparidad. También se muestra una superposición de la imagen original y el mapa de disparidades en color para una visualización más intuitiva al ojo humano.

a) Imagen original

b) Ground truth



c) Mapa de disparidades método CSBP

d) Imagen original superpuesta al mapa de disparidades en gradiente de color

Figura 51: Imágenes relevantes en la comparativa del error de la estimación a) Imagen original obtenida de la base de datos KITTI b) Ground truth correspondiente c) Mapa de disparidades con el método CSBP d) Mapa de disparidades en gradiente de color superpuesto a imagen original

9.1.1. Resultados con código ejecutado en Jetson TK1

	Tiempo transcurrido [ms]		Error de disparidad [%]	
	Media	Mediana	Media	Mediana
BM	156.42	153.33	55.76%	55.60%
SGBM	3515.28	3512.92	23.54%	22.64%
CUDA_BM	483.01	498.78	30.64%	30.13%
CUDA_BP	5085.87	5061.72	32.56%	32.68%
CUDA_CSBP	1050.84	1041.92	21.81%	20.33%
CUDA_BP_estimado	13084.36	13067.80	32.50%	32.56%
CUDA_CSBP_estimado	680.04	674.13	23.08%	21.71%

Tabla 1: Resultados obtenidos de la función del cálculo del error de disparidad en la Jetson

	E_medio		E_mediana		Densidad de pixeles	
	Media	Mediana	Media	Mediana	Media	Mediana
BM	0.89	0.80	0.54	0.53	42.04%	41.95%
SGBM	1.57	1.36	0.72	0.67	75.93%	77.32%
CUDA_BM	4.48	4.14	1.30	1.24	77.40%	79.19%
CUDA_BP	7.07	6.47	1.67	1.43	99.83%	99.83%
CUDA_CSBP	4.40	3.68	1.04	0.95	99.84%	99.84%
CUDA_BP_estimado	6.88	6.24	1.68	1.44	99.82%	99.82%
CUDA_CSBP_estimado	4.86	4.11	1.10	1.00	99.86%	99.86%

Tabla 2: Resultados obtenidos de la función del cálculo del error de disparidad en la Jetson

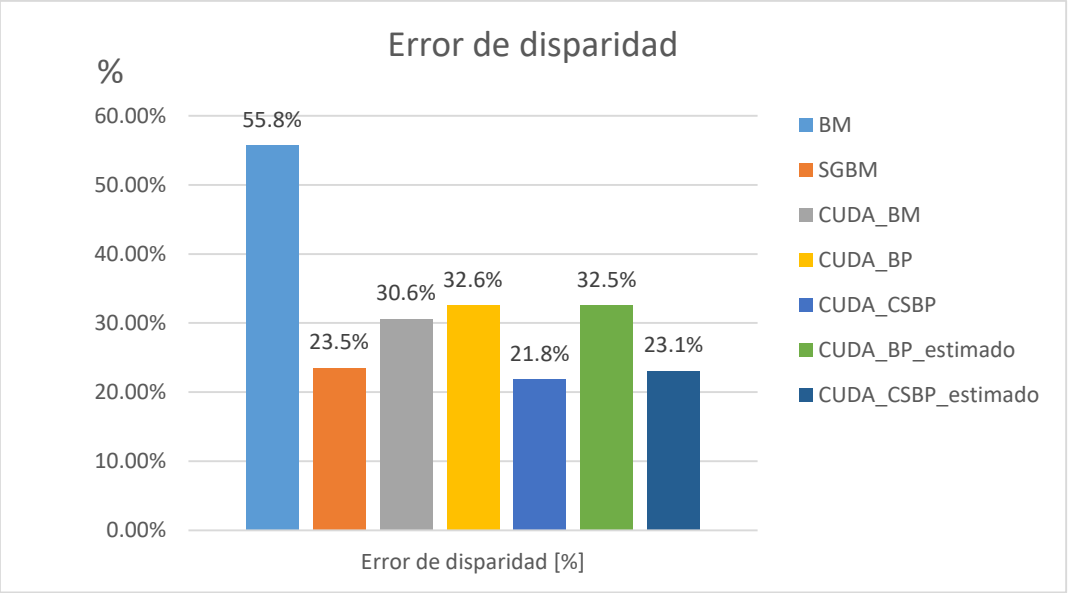


Figura 52: Resultados del cálculo del error de disparidad en Jetson TK1

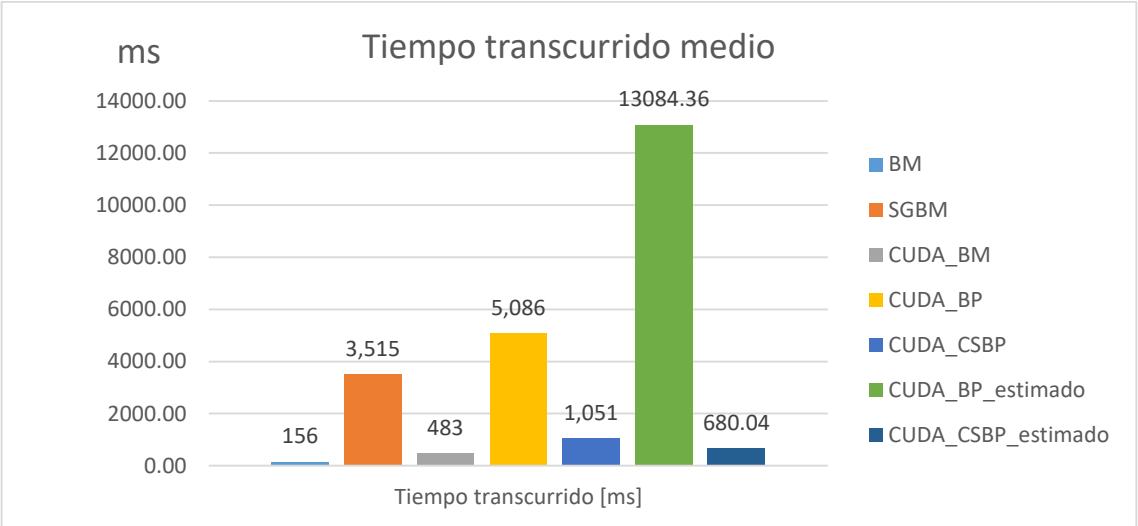


Figura 53: Resultados del tiempo transcurrido en el algoritmo correspondiente ejecutados en Jetson TK1

9.1.2. Resultados con código ejecutado en portátil ASUS

Se muestran los resultados mismo resultados vistos anteriormente, pero ejecutados en el portátil usado en gran parte de este trabajo, simplemente para tener una referencia con la que comparar los resultados de la Jetson TK1.

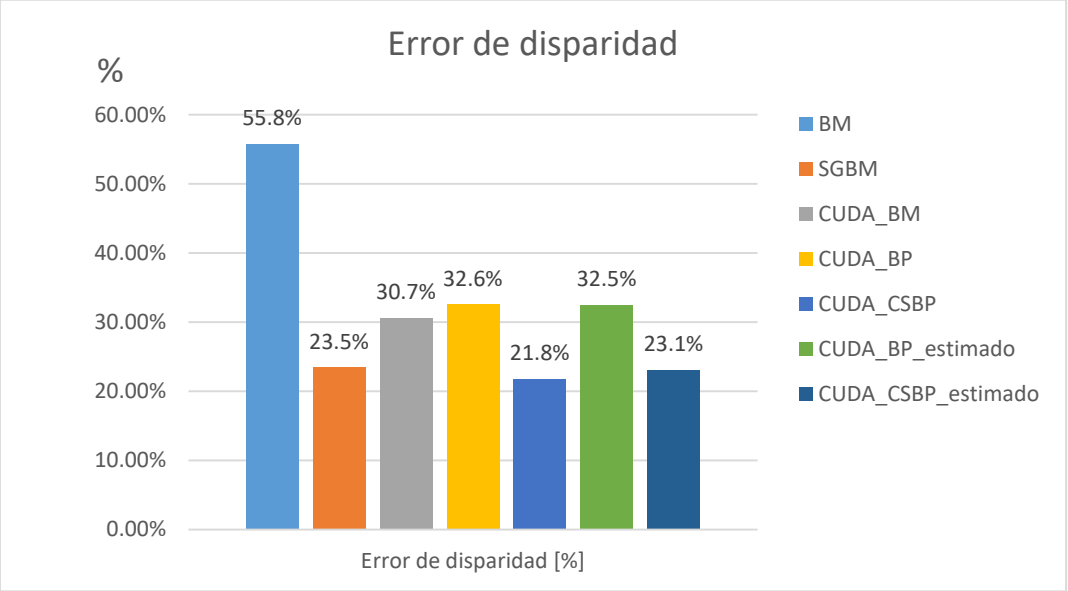


Figura 54: Resultados del cálculo del error de disparidad en portátil ASUS

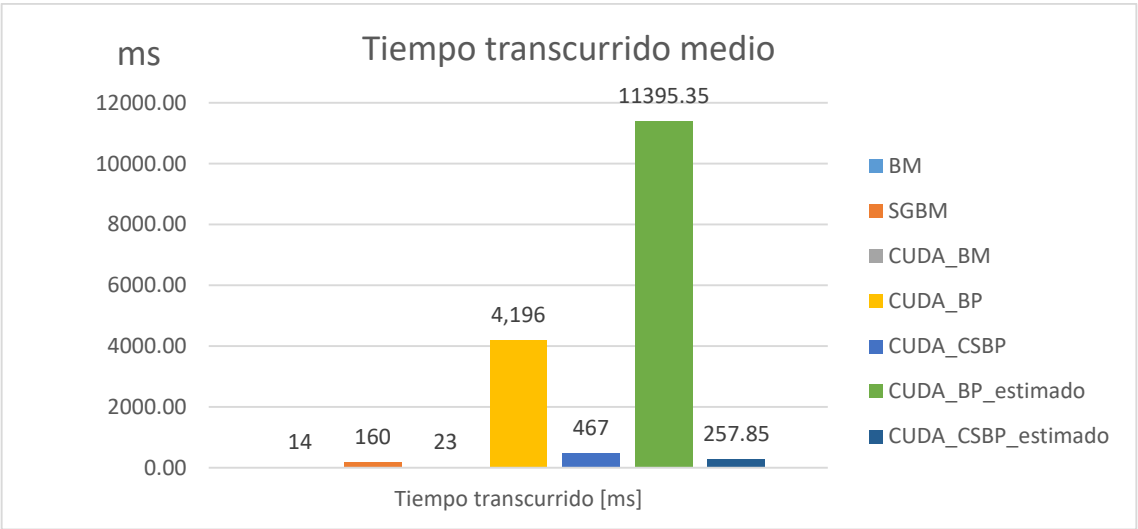


Figura 55: Resultados del tiempo transcurrido en el algoritmo correspondiente ejecutados en ASUS

	Tiempo transcurrido [ms]		Error de disparidad [%]	
	Media	Mediana	Media	Mediana
BM	14.39	13.97	55.77%	55.61%
SGBM	160.47	159.43	23.45%	22.63%
CUDA_BM	22.73	22.82	30.65%	30.13%
CUDA_BP	4196.03	4195.38	32.56%	32.68%
CUDA_CSBP	467.15	459.67	21.81%	20.33%
CUDA_BP_estimado	11395.35	11399.70	32.50%	32.56%
CUDA_CSBP_estimado	257.85	256.41	23.08%	21.76%

Tabla 3: Resultados obtenidos con la función de cálculo del error de disparidad en portátil ASUS.

	E_media		E_mediana		Densidad de pixeles	
	Media	Mediana	Media	Mediana	Media	Mediana
BM	0.88	0.79	0.54	0.53	41.98%	41.94%
SGBM	1.69	1.41	0.73	0.68	76.77%	78.13%
CUDA_BM	4.48	4.14	1.30	1.24	77.38%	79.20%
CUDA_BP	7.07	6.47	1.67	1.43	99.84%	99.85%
CUDA_CSBP	4.40	3.68	1.04	0.95	99.87%	99.88%
CUDA_BP_estimado	6.88	6.24	1.68	1.44	99.83%	99.84%
CUDA_CSBP_estimado	4.86	4.09	1.10	1.00	99.86%	99.87%

Tabla 4: Resultados obtenidos con la función de cálculo del error de disparidad en portátil ASUS.

9.2. Reconstrucción final

Para la reconstrucción final, se utilizan secuencias grabadas con la cámara estéreo Bumblebee montada en el vehículo IVVI, las cuales tienen una dimensión de 1280x960 pixeles, a diferencia de las imágenes usadas de la base de datos KITTI. Las siguientes reconstrucciones expuestas han sido calculadas y visualizadas en el portátil ASUS, debido a la imposibilidad de visualizar en la Jetson TK1 las nubes de puntos al mismo tiempo que estas se generan. Este proceso conlleva una gran carga computacional para la cual la Jetson no tiene los recursos necesarios.

En la Figura 56 se muestra una reconstrucción de alta densidad de una secuencia grabada con el IVVI por un entorno urbano.

a) Punto de vista desde el inicio de la secuencia

b) Punto de vista dentro de la nube

c) Punto de vista dentro de la nube

Figura 56: Reconstrucción de una misma secuencia grabada con el IVVI y calculada con el método CSBP. Reproducida a x0.3 de la velocidad de reproducción de la secuencia original.

En la Figura 57 y Figura 58 se visualiza una reconstrucción hecha con el algoritmo CSBP y la creación de nubes de puntos en GPU. En este caso, se han ejecutados dos filtrados a la imagen resultante. El primero, es el comentado en la sección 5 voxel_grid. Éste reduce la cantidad de la nube de puntos y, además, la limita en el eje Z (de 2 a 30 metros en este caso). De esta forma evitamos que aparezcan puntos muy cercanos, como puede ser el capó del coche, y puntos muy lejanos en donde la precisión es menor. El segundo filtrado es el nodo de ROS Passthrough_filter. Es necesario este nodo extra porque como se puede observar en la Figura 59, debido a regiones sin textura como el cielo en este caso, el algoritmo CSBP produce un error y estima erróneamente la distancia del cielo.

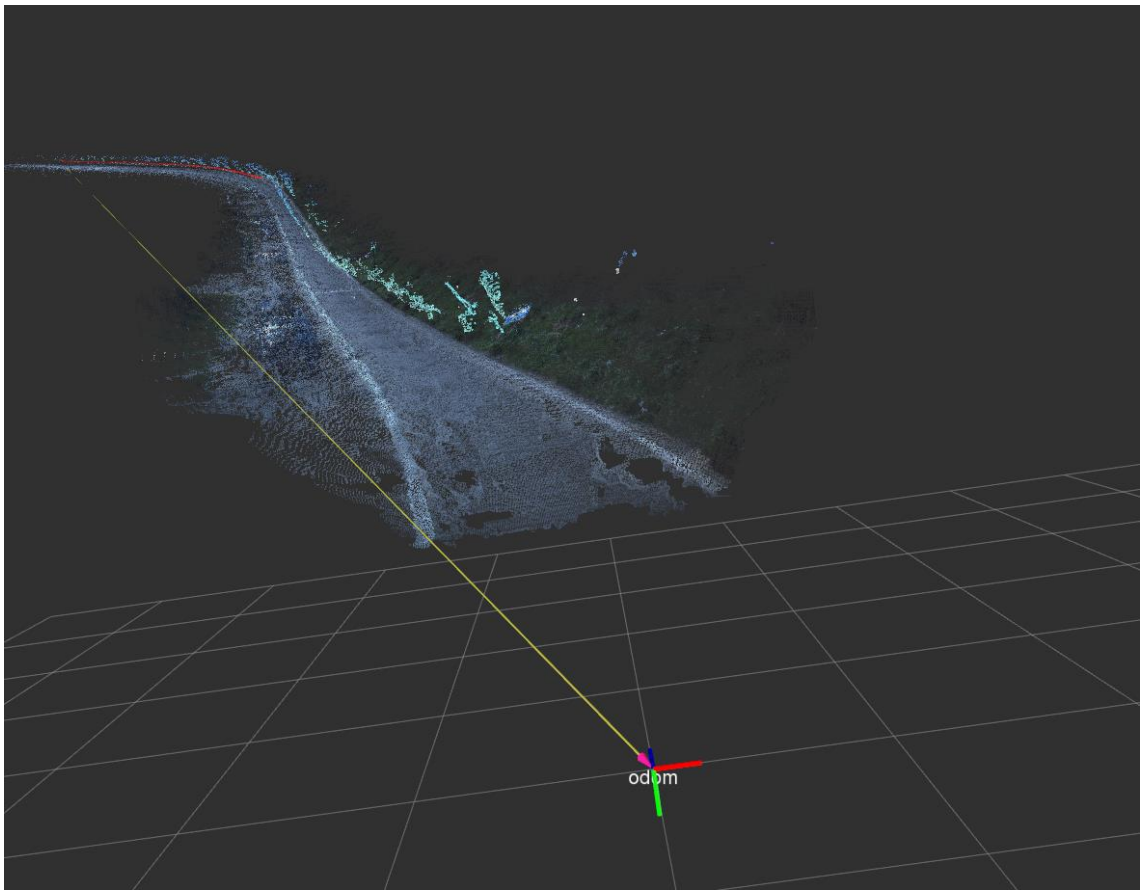


Figura 57: Reconstrucción a x0.2 velocidad de reproducción

Una reconstrucción completa a tiempo real con el método CSBP no es posible con esta resolución de imagen (1280x960) y a esa velocidad de desplazamiento. A la mitad de la velocidad de reproducción de las imágenes guardadas en una secuencia real grabada con el IVVI, se obtiene una nube de puntos continua, como se muestra en la Figura 60, aunque ya se aprecia que la tasa de creación del mapa de disparidades comienza a limitar el funcionamiento. Más allá de esa velocidad de reproducción la nube de puntos queda discontinua, como se aprecia en la Figura 61, donde el archivo de video (.bag) se reproduce a tiempo real. En la Figura 62 se puede observare que cuanto más despacio se reproduzca el bag (o más despacio se desplace el vehículo/robot en cuestión) más densa será la reconstrucción resultante.

Si cambiamos el algoritmo del cálculo del mapa de disparidad al método StereoBM implementado en CUDA, el cual es aproximadamente el doble de rápido que el CSBP, la limitación en la reconstrucción a tiempo real no estaría en el algoritmo del mapa de disparidad, si no en la odometría. Es importante destacar que el foco de este proyecto se centraba en la integración de un sistema de reconstrucción tridimensional con vistas a su uso en la Jetson, aprovechando sus capacidades GPGPU. El cálculo de la odometría no formaba parte de los objetivos iniciales, y por esto se usó un paquete de ROS encargado de realizar esta tarea.

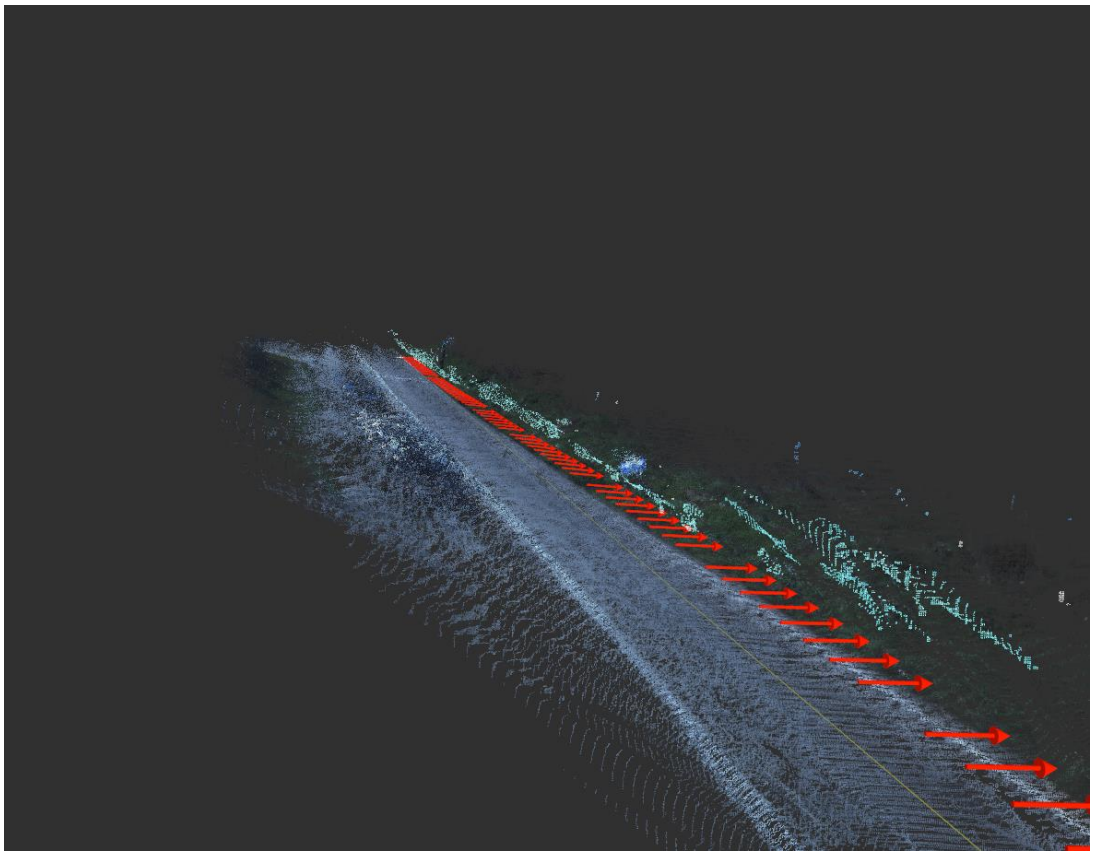


Figura 58: Reconstrucción a x0.2 velocidad de reproducción

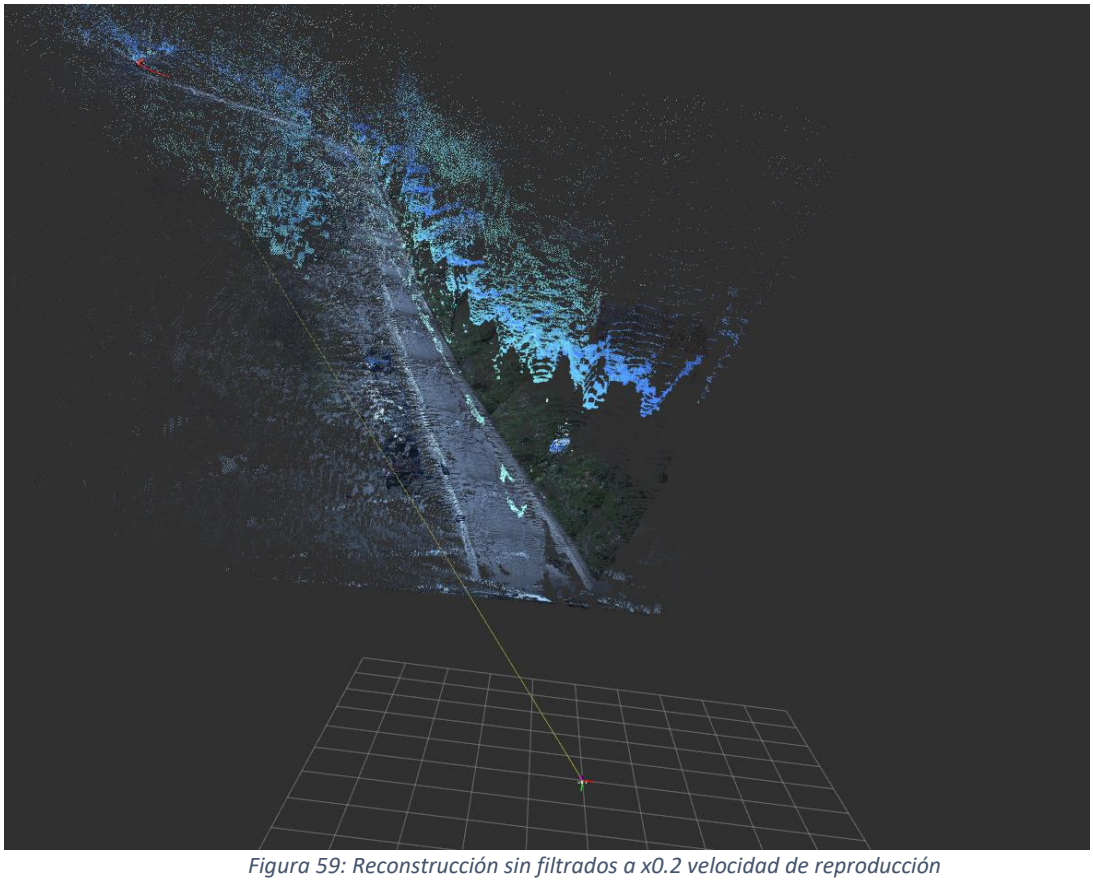


Figura 59: Reconstrucción sin filtrados a x0.2 velocidad de reproducción

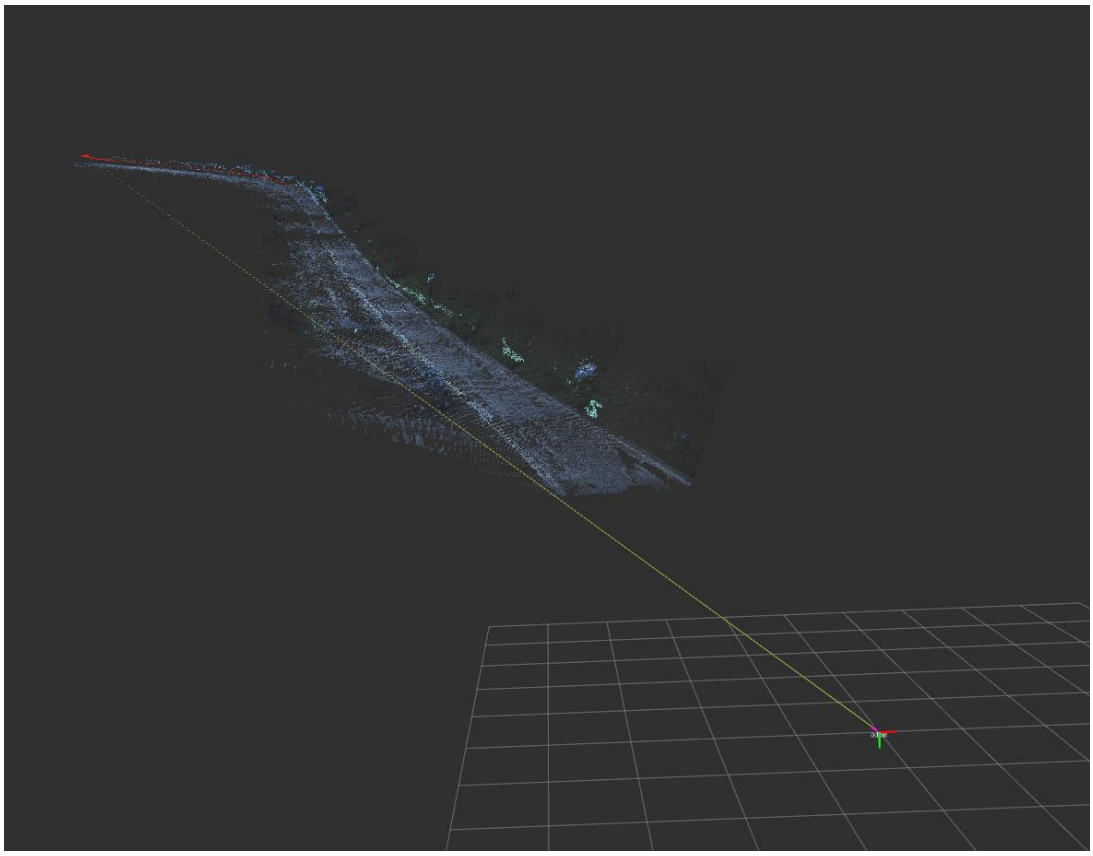


Figura 60: Reconstrucción a x0.5 velocidad de reproducción

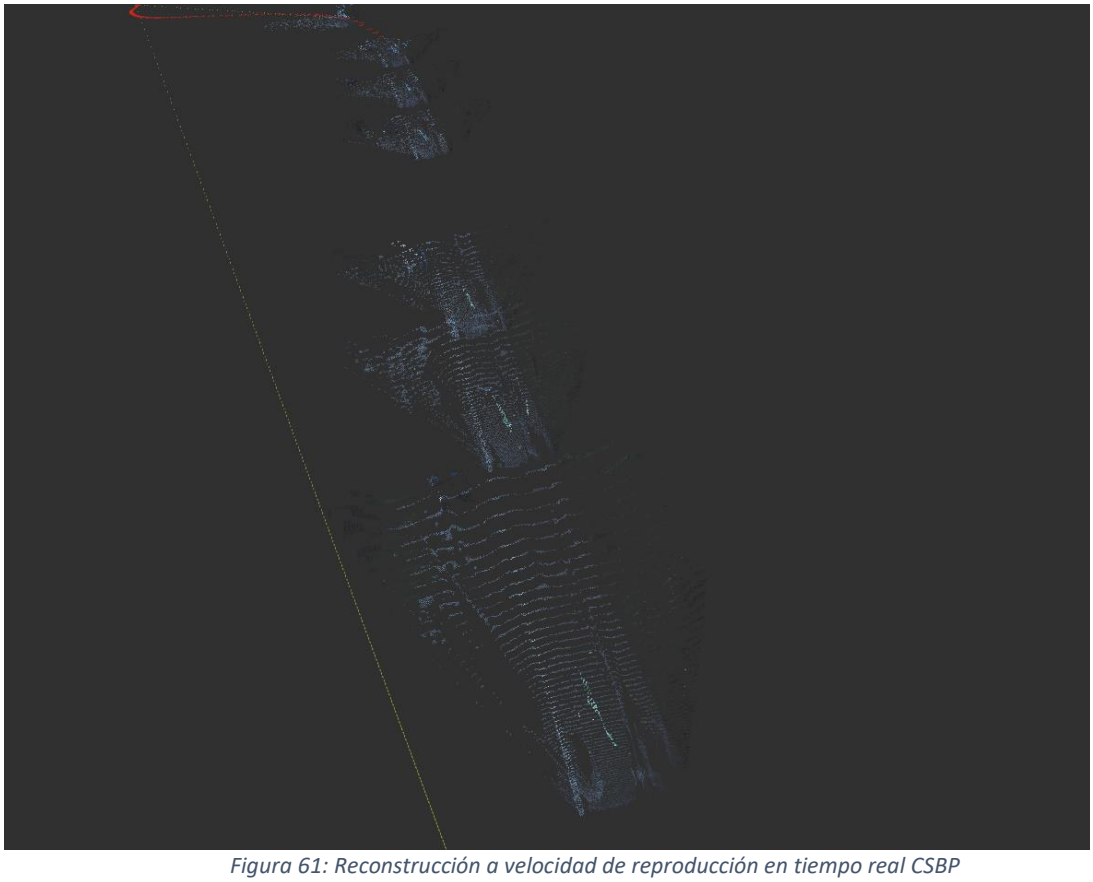


Figura 61: Reconstrucción a velocidad de reproducción en tiempo real CSBP

Figura 62: Reconstrucción a x0.1 velocidad de reproducción

Aunque la visualización de las reconstrucciones arriba mostradas fue realizada en el portátil ASUS debido a la falta de recursos de la Jetson para realizar ambos procesos reconstrucción y visualización, si se realizaron pruebas de la tasa de generación de nubes de puntos con ambos algoritmos BM y CSBP, así como con el proceso ejecutado en CPU con el método SGBM y generación de nubes de puntos con las herramientas de PCL nativas de ROS. Gracias al comando `rostopic hz` de ROS, el cual ofrece la frecuencia de publicación de un topic, se obtuvieron los siguientes resultados.

Frecuencia publicación	GPU BM (Hz)	GPU CSBP (Hz)	SGBM (Hz)
Bag	8.53	7.02	7.16
Mapa de disparidades	1.41	0.32	0.100
Nube de puntos	0.51	0.23	-

Tabla 5: Resultados de la frecuencia de publicación de mapas de disparidad y nubes de puntos en la Jetson

La frecuencia de publicación de la secuencia (Bag en la tabla de arriba), debería estar en torno a 10 Hz o 10 fotogramas por segundo. Es posible que, debido a las limitadas capacidades de la CPU en la Jetson, la secuencia no se pudiera reproducir a tiempo real, es decir, simulando que las imágenes fuesen captadas directamente de la cámara.

En la Figura 63, Figura 64 y Figura 65 se muestran los resultados de calcular la frecuencia de generación de nube de puntos en la Jetson, resumidos en la Tabla 5. En el caso del método SGBM, no se pudo obtener una frecuencia de generación de nubes de puntos, se muestra solo la frecuencia de generación del mapa de disparidades. Se cree que la carga computacional para la CPU era demasiado grande.

```
no new messages
average rate: 0.459
        min: 1.745s max: 2.715s std dev: 0.40263s window: 4
average rate: 0.482
        min: 1.745s max: 2.715s std dev: 0.39149s window: 5
no new messages
average rate: 0.532
        min: 1.095s max: 2.715s std dev: 0.52566s window: 6
no new messages
average rate: 0.530
        min: 1.095s max: 2.715s std dev: 0.48004s window: 7
average rate: 0.524
        min: 1.095s max: 2.715s std dev: 0.44795s window: 8
no new messages
no new messages
average rate: 0.512
        min: 1.095s max: 2.715s std dev: 0.43645s window: 9
average rate: 0.517
        min: 1.095s max: 2.715s std dev: 0.41507s window: 10
no new messages
average rate: 0.526
        min: 1.095s max: 2.715s std dev: 0.40618s window: 11
no new messages
average rate: 0.524
```

Figura 63: Frecuencia de generación de nubes de puntos en Jetson TK1 con el método BM

```
average rate: 0.205
        min: 1.837s max: 8.395s std dev: 3.01170s window: 5
no new messages
no new messages
average rate: 0.226
        min: 1.837s max: 8.395s std dev: 2.85121s window: 6
no new messages
average rate: 0.242
        min: 1.837s max: 8.395s std dev: 2.67702s window: 7
no new messages
no new messages
no new messages
no new messages
no new messages
average rate: 0.228
        min: 1.837s max: 8.395s std dev: 2.54619s window: 8
no new messages
no new messages
average rate: 0.240
        min: 1.837s max: 8.395s std dev: 2.44992s window: 9
no new messages
no new messages
average rate: 0.248
        min: 1.837s max: 8.395s std dev: 2.34121s window: 10
```

Figura 64: Frecuencia de generación de nubes de puntos en Jetson TK1 con el método CSBP

```
average rate: 0.098
        min: 10.252s max: 10.259s std dev: 0.00386s window: 3
no new messages
no new messages
no new messages
no new messages
no new messages
no new messages
no new messages
no new messages
no new messages
average rate: 0.100
        min: 9.589s max: 10.259s std dev: 0.31415s window: 4
no new messages
no new messages
no new messages
no new messages
no new messages
no new messages
no new messages
no new messages
no new messages
average rate: 0.100
        min: 9.589s max: 10.259s std dev: 0.27304s window: 5
```

Figura 65: Frecuencia de generación de mapa de disparidades en Jetson TK1 con el método SGBM. No fue posible obtener nubes de puntos.

10. Presupuesto

Para la realización del cálculo del presupuesto se evaluarán el número de horas de trabajo del ingeniero, y el valor uso de los componentes o su desamortización.

El proyecto ha tenido una duración aproximada de 5 meses con una media de unas 6h diarias trabajadas, sumando un total de aproximadamente 600 horas. Este será el tiempo utilizado para calcular el sueldo del ingeniero y de las desamortizaciones.

Para el cálculo del coste del ingeniero se utiliza como base salarial 1.600€ mensuales, o 19.200€ brutos anuales.

$$Coste = 1600\frac{€}{mes} * \frac{6h}{8h} = 1200\frac{€}{mes}$$

Ingeniero	Tiempo dedicado (meses)	Sueldo(anual)	Coste total (€)
Diego Cabo Golvano	5	19,200 €	6,000 €

Tabla 6: Coste del personal

Para el cálculo de la desamortización del material se han usado las tablas de amortizaciones publicadas por el ministerio de Hacienda [33], donde se determina que los equipos electrónicos tienen una amortización del 20% y los equipos de video del 33%.

Además, se ha estimado un porcentaje de uso para conseguir una desamortización más realista, dado que el uso no ha sido el mismo con los diferentes aparatos (por ejemplo, la cámara solo fue usada al final con un par de grabaciones).

$$Precio\ de\ desamortización = Precio * Uso * coeficiente\ de\ amortizacion * \frac{5\ meses}{12\ meses}$$

Equipamiento	Precio	Porcentaje de uso	Coeficiente de amortizacion	Precio de desamortizacion
Portatil ASUS	749 €	100.00%	0.2	62.42 €
Jetson TK1	153.31 €	35.00%	0.2	4.47 €
Cámara Bumblebee	3,500 €	3.00%	0.33	14.44 €

Tabla 7: Costes materiales

Materiales	81.33 €
Personal	6,000.00 €
TOTAL	**6,081.33 €**

Tabla 8: Costes totales

11.Conclusiones

La realización de este proyecto ha supuesto un gran reto en varios aspectos. A nivel técnico, se han pasado decenas de horas estudiando nuevos conceptos, como visión estéreo o nubes de puntos, tecnologías que no habían sido vistas antes como puede ser el meta sistema operativo ROS, o la puesta a punto de los equipos con todas las librerías y programas necesarios para la realización de este proyecto. A nivel personal, ha requerido de una gran fuerza de voluntad y paciencia, ya que una cantidad innumerable de horas se han gastado en la comprensión de todos los conceptos técnicos y frente al ordenador intentando depurar el programa, o simplemente averiguar el motivo de que no compilase.

En cuanto a los resultados finales la sensación es bastante satisfactoria. Se ha conseguido crear un sistema que podría ser implementado en un pequeño contenedor (sensor óptico, conexiones y microprocesador todo incluido) y que puede ser utilizado como instrumento complementario a la navegación espacial de una maquina autónoma. Se consigue un sistema que se acerca al funcionamiento en tiempo real, con una precisión muy alta. Aunque se llega a acercar al tiempo real, este sistema tiene varias limitaciones que impiden conseguir una reconstrucción en tiempo real completamente. En primer lugar, como se comentó anteriormente, el cálculo de la odometría usado en este sistema tiene problemas para alcanzar un modo de funcionamiento en tiempo real cuando el vehículo se desplaza a cierta velocidad. En segundo lugar, el método CSBP para calcular las disparidades tampoco consigue tiempos de procesamiento aceptables para tiempo real cuando el vehículo o robot se desplaza a una cierta velocidad. Habría que recurrir al método BM e incluso en ese caso no se garantiza una reconstrucción densa a tiempo real.

No obstante, uno de los principales objetivos de este proyecto era el de demostrar la mejora en la eficiencia de estos algoritmos de visión estéreo ejecutados en la GPU del microcontrolador Jetson TK1. Como se vio en la sección 9.2, la mejora en la ejecución del método CSBP en la GPU con respecto al método en CPU SGBM es notable. Se obtienen mapas de disparidad a una frecuencia en torno a 3 veces superior, y se llega a generar una nube de puntos, lo cual utilizando el método SGBM y la creación de nube de puntos en CPU no es posible.

Gracias a la filosofía “open source” en la que se basa ROS, se ha podido crear una aplicación completa desde el nivel de píxel hasta el resultado final en un tiempo limitado, a cambio de no tener todo el control sobre cómo funciona sistema (por ejemplo, la odometría). Además, debido a las características de ROS, se consigue una gran flexibilidad en este sistema. Algunas de las ventajas de esta flexibilidad pueden ser:

I. Cualquiera de estos nodos puede ser modificado y mejorado individualmente sin la necesidad de tener conocimientos sobre el resto del sistema. Por ejemplo, se podría mejorar el cálculo de la odometría utilizando otras técnicas como puede ser la odometría de las ruedas.
II. Se pueda utilizar un sensor diferente al utilizado en este proyecto por requerimientos del usuario. El sensor Bumblebee es relativamente caro y pesado, y se convierte con diferencia, en la parte más costosa de este sistema.

Por ello, algunos clientes con diferentes necesidades podrían buscar un sistema más ligero, o más barato. Con pocas modificaciones en el sistema, se podría utilizar un sensor más pequeño, más ligero, o más barato.

Como se vio en la sección 1.1.2, la introducción de este tipo de sensores a gran escala reducirá los costes de una cámara estéreo de alta calidad, permitiendo en un futuro próximo la normalización en el uso de sensores de calidad similar al utilizado en este proyecto.

Se espera que este sistema aquí propuesto pueda incorporarse al IVVI como funcionalidad con la que no contaba hasta el momento.

12. Futuras mejoras

Durante el transcurso de este proyecto han ido surgiendo dudas o problemas, en donde se ha tenido que tomar decisiones basadas en las restricciones de tiempo para este proyecto. También han ido surgiendo ideas que no se han realizado por la falta de tiempo y que podrían mejorar la robustez y fiabilidad del programa. A continuación, se comentan algunos de estas posibles mejoras.

Un paso hacia la mejora en la robustez de este programa, sería el de realizar una comprobación de la existencia de GPU en el sistema, y si esta es capaz de ejecutar los algoritmos de CUDA presentados. Esta operación adquiriría información sobre la potencia del hardware grafico en cuestión y esta información podría ser utilizada para decidir sobre que método de cálculo de disparidad utilizar, y conseguir así un algoritmo adaptativo a las capacidades del hardware.

En cuanto a la mejora de los algoritmos, se podrían realizar post-procesados al mapa de disparidades, como aplicación de filtros, que conseguirían un mapa de disparidades con menos ruido, eliminando posibles errores en zonas aisladas debidos a problemas de iluminación o regiones sin textura.

Otra futura mejora sería la de implementar algún tipo de detector o clasificador que identificase los objetos en movimiento, como pueden ser los demás vehículos en circulación, y posteriormente eliminarlos. Así se conseguirían mapas tridimensionales precisos independientemente de que en las grabaciones previas hubiese oclusiones con objetos en movimiento.

Se podría intentar mejorar el código de las librerías pcl en donde se genera la nube de puntos. En este proyecto ya se modificó ligeramente, pero mientras se realizaba se descubrieron nuevos problemas. Una característica importante es que las matrices de las imágenes de disparidad y de color se pasan a la función con un puntero de tipo *int*. Este puntero de tipo *int* implica que la futura nube de puntos generada va a ser discontinua (hay un salto entre números, por ejemplo, de 4 se pasaría a 5). De ahí que veamos franjas oscuras o sin representación en la reconstrucción. Se debería estudiar esa clase, y ver si es posible cambiar estos punteros en la llamada de la función a tipo *float.*

13. Bibliografía

[1] «Informe mundial sobre prevención de los traumatismos causados por el tránsito,» Organizacion Mundial de la Salud, Ginebra, 2004.

[2] «Las principales cifras de la siniestralidad vial en España 2014,» Dirección General de Tráfico, Madrid, 2014.

[3] R. Bagat, «Here's What You Need To Know About Google's Project Tango,» *Forbes,* 20 Jun 2016.

[4] «Global 3D Camera Market Size, Share, Development, Growth and Demand Forecast to 2022,» P&S Market Research, 2016.

[5] «3D Camera Market by Type, Technology and Application - Global Opportunity Analysis and Industry Forecast, 2014 - 2020,» Allied Market Research, 2016.

[6] Z. Epstein, «Kinect sales,» BGR, 12 Febrero 2013. [En línea]. Available: http://bgr.com/2013/02/12/microsoft-xbox-360-sales-2013-325481/.

[7] «Kinect confirmed as fastest-selling consumer electronics device,» Guinness world records, [En línea]. Available: http://community.guinnessworldrecords.com/_Kinect-Confirmed-As-Fastest-Selling-Consumer-Electronics-Device/blog/3376939/7691.html.

[8] Bubba, «Quality logo products,» [En línea]. Available: https://www.qualitylogoproducts.com/blog/microsoft-kinect-fastest-selling-consumer-electronic/.

[9] A. J. Barry y R. Tedrake, «Pushbroom stereo for high-speed navigation in cluttered environments,» de *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, 2015.

[10] D. Shapiro, «Beyond GPS: How HD Maps Will Show Self-Driving Cars the Way,» Blogs Nvidia, 5 Abril 2016. [En línea]. Available: https://blogs.nvidia.com/blog/2016/04/05/self-driving-cars-2/.

[11] P. Henry, M. Krainin, E. Herbst, X. Ren y D. Fox, «RGB-D Mapping: Using Depth Cameras for Dense 3D Modeling of Indoor Environments,» de *Experimental Robotics*, vol. 79, Springer, pp. 477-491.

[12] T. Zhang y Y. Nakamura, «Surface Reconstruction with Sparse Point Clouds of Velodyne Sensor,» de *The 14th IFToMM World Congress*, Taipei, Taiwan, 2015.

[13] A. Degirmenci, P. M. Loschak, C. M. Tschabrunn, E. Anter y R. D. Howe, «Compensation for Unconstrained Catheter Shaft Motion,» de *2016 IEEE International Conference on Robotics and Automation (ICRA)*, Stockholm, Sweden, 2016.

[14] «Advanced Real-Time Visualization for Robotic Heart Surgery,» News developer Nvidia, 12 Agosto 2016. [En línea]. Available: https://news.developer.nvidia.com/advanced-real-time-visualization-for-robotic-heart-surgery/.

[15] A. de la Escalera Hueso, Visión por computador: fundamentos y métodos, Prentice Hall, 2001.

[16] R. Szeliski, «Chapter 11 Stereo correspondence,» de *Computer Vision: Algorithms and Applications*, Springer, 2010, pp. 537-571.

[17] «Optical Flow,» OpenCV 3.1 docs, [En línea]. Available: http://docs.opencv.org/trunk/d7/d8b/tutorial_py_lucas_kanade.html.

[18] C. Loop y Z. Zhang, «Computing rectifying homographies for stereo vision,» de *Computer Vision and Pattern Recognition, 1999. IEEE Computer Society Conference on*, Fort Collins, CO, 1999.

[19] Tosaka, «CUDA,» Wikipedia, 29 Julio 2016. [En línea]. Available: https://en.wikipedia.org/wiki/CUDA.

[20] elinux.org, «Jetson TK1,» 11 Agosto 2016. [En línea]. Available: http://elinux.org/Jetson_TK1.

[21] «Research- ADAS,» UC3M, 2016. [En línea]. Available: http://portal.uc3m.es/portal/page/portal/dpto_ing_sistemas_automatica/investigacion/IntelligentSystemsLab/research/adas#IVVI2.

[22] «About us,» QT creator, [En línea]. Available: https://www.qt.io/about-us/.

[23] «About,» OpenCV, [En línea]. Available: http://opencv.org/about.html.

[24] «About us,» pcl, [En línea]. Available: http://pointclouds.org/about/.

[25] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler y A. Ng, «ROS: an open-source Robot Operating System,» *ICRA workshop on open source software,* vol. 3, nº 3.2, p. 5, 2009.

[26] «Voxel_grid,» ROS.org, [En línea]. Available: http://wiki.ros.org/voxel_grid.

[27] docs.opencv.org, «Camera Calibration and 3D Reconstruction,» [En línea]. Available: http://docs.opencv.org/3.1.0/d9/d0c/group__calib3d.html#gsc.tab=0.

[28] P. F. Felzenszwalb y D. P. Huttenlocher, «Efficient Belief Propagation for Early Vision,» *International Journal of Computer Vision,* vol. 70, nº 1, pp. 45-54, 2006.

[29] Q. Yang, L. Wang y N. Ahuja, «A constant-space belief propagation algorithm for stereo matching,» de *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, San Francisco, CA, 2010.

[30] H. Hirschmuller, «Stereo Processing by Semiglobal Matching and Mutual Information,» *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 30, nº 2, pp. 328-341, 2008.

[31] A. Geiger, P. Lenz y R. Urtasun, «Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite,» de *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[32] S. Kohlbrecher y J. Meyer, «hector_slam,» ROS.org, [En línea]. Available: http://wiki.ros.org/hector_slam/Tutorials/SettingUpForYourRobot.

[33] «BOE - Tablas de amortización,» Ministerio de Hacienda, 2015. [En línea]. Available: http://www.boe.es/buscar/act.php?id=BOE-A-2004-14600.

[34] A. J. Z. C. S. Geiger, «Stereoscan: Dense 3d reconstruction in real-time,» de *Intelligent Vehicles Symposium (IV)*, vol. IV, IEEE, 2011, pp. 963-968.